USENIX

MICRO-KERNELS and OTHER KERNEL ARCHITECTURES PROCEEDINGS

SPRING 1992

U

US

USE

U NIX

USENIX

WORKSHOP PROCEEDINGS

Micro-kernels and Other Kernel Architectures

April 27 - 28, 1992
Seattle, Washington

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 U.S.A.

The price is $30 for members and $39 for non-members.

Outside the U.S.A and Canada, please add
$20 per copy for postage (via air printed matter).

ISBN 1-880446-42-1

Printed in the United States of America on 50% recycled paper, 10-15% post-consumer waste.

Proceedings of the

USENIX Workshop
on
Micro-Kernels and Other Kernel Architectures

USENIX Association

April 27 - 28, 1992
Seattle, Washington

Program
Micro-kernels and Other Kernel Architectures Workshop

Seattle, Washington
April 27 - 28, 1992

Monday, April 27

8:30 - 8:35	Opening Remarks *Lori Grob, Chorus systèmes*
8:35 - 9:35	Amoeba .... 1 *Robbert van Renesse, Vrije University/Cornell University*
9:35 - 10:35	Mach .... 11 *Richard Draves, Carnegie Mellon University*
11:05 - 12:05	Plan 9 .... 31 *David Presotto, AT&T Bell Laboratories*
1:30 - 2:30	Chorus .... 39 *Marc Rozier, Chorus systèmes.*
2:30 - 3:30	NT (Paper not included in this proceedings) *David Cutler, Microsoft Corp.*
4:00 - 5:00	Micro-kernels Panel Session: What is a Micro-kernel? Are There Really Any Differences? *Moderator: Peter Honeyman, CITI University of Michigan*
5:00 - 6:00	**New Architectures** Chair: *TBA*
	Modularity and Interfaces in Micro-kernel Design and Implementation: A Case Study of Chorus on the HP PA-Risc .... 71 *Jonathan Walpole, Jon Inouye, Ravindranath Konuru, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology*
	A Micro-Kernel Architecture for Next Generation Processor .... 83 *Toshio Okamoto, Hideo Segawa, Sung Ho Shin, Hiroshi Nozue, Ken-ichi Maeda, Mitsuo Saito, Toshiba Corporation Research and Development Center Information Systems Lab.*

Tuesday, April 28

8:30 - 10:30	**New Systems** Chair: *Robbert van Renesse*
	The KeyKOS® Nanokernel Architecture .... 95 *Allen C. Bomberger, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, Jonathan S. Shapiro, Key Logic Corp.*
	An Architectural Overview of QNX .... 113 *Dan Hildebrand, Quantum*
	An Architectural Overview of Alpha: A Real-Time, Distributed Kernel .... 127 *Franklin Reynolds, Open Software Foundation*
	Performance of the BirLiX Operating System .... 147 *P. Schüller, H. Härtig, W.E. Kühnhauser, German National Research Center For Computer Science (GMD)*

11:00 - 12:30 **Lessons Learned**
Chair: *Edward Lazowska*

Multimedia/Realtime Extensions for Mach 3.0 161
Jun Nakajima, Masatomo Yazaki, Hitoshi Matsumoto, Human Interface Laboratory #1, Fujitsu Laboratories Ltd.

Reimplementing the Synthesis Kernel 177
Henry Massalin, Calton Pu, Dept. of Computer Science, Columbia University

A Model and Prototype of VMS Using the Mach 3.0 Kernel 187
Cheryl A. Wiecek, Digital Equipment Corporation

2:00 - 3:30 **Experience and Observations I**
Chair: *Lori S. Grob*

The Increasing Irrelevance of IPC Performance for Micro-kernel-Based Operating Systems 205
Brian Bershad, School of Computer Science, Carnegie Mellon University

Fast Thread Management and Communication Without Continuations 213
Jochen Liedtke, German National Research Center for Computer Science (GMD)

Experience with SVR4 Over Chorus 223
Nariman Batlivala, Barry Gleeson, Jim Hamrick, Scott Lurndal, Darren Price, James Soddy, Unisys Corporation; Vadim Abrossimov, Chorus systèmes

4:00 - 6:00 **Experience and Observations II**
Chair: *Avadis Tevanian*

Data Movement in Kernelized Systems 243
Randall W. Dean, Francois Armand, CMU, Chorus systèmes

Distributed Abstractions, Lightweight References 263
Marc Shapiro, Mesaac Makpangou, INRIA

Reliable Multicast between Micro-Kernels 269
Robbert van Renesse, Ken Birman, Robert Cooper, Bradford Glade, Patrick Stephenson, Cornell University

Designing a Scalable Operating System for Shared Memory Multiprocessors..........285
Michael Stumm, Ronald Unrau, and Orran Krieger, Department of Electrical Engineering, University of Toronto

Program Committee:

Program Chair
Lori S. Grob, Chorus systemes

Edward D. Lazowska, University of Washington
Robbert van Renesse, Cornell University/Vrije Universiteit
Avadis Tevanian, Jr., NeXT Computer, Inc.

Short Overview of Amoeba

Robbert van Renesse
<rvr@cs.cornell.edu>
Dept. of Computer Science
4118 Upson Hall
Cornell University
Ithaca, NY 14853

Andrew S. Tanenbaum
<ast@cs.vu.nl>
Vrije Universiteit
Faculteit Wiskunde en Informatica
De Boelelaan 1081
Amsterdam, The Netherlands

ABSTRACT

This paper presents a short overview of the Amoeba distributed operating system, focusing in particular on its microkernel for tutorial purposes. The Amoeba system uses the popular object-based model for distributed computing, implemented using remote procedure calls and lightweight processes. Amoeba consists on a set of microkernels, with objects implemented by user services. Amoeba was designed to be used, and therefore we have devoted considerable energy to performance. We report on the implementation and performance of Amoeba remote procedure call.

1. Architecture of the Amoeba System

Amoeba is an object-based distributed system that is being developed at the Vrije Universiteit in Amsterdam. Amoeba currently runs on Motorola 68020, Intel 80386, MicroVax II, and SPARC processors. Several networks are supported by Amoeba, among others Ethernet and the Pronet token ring. Amoeba does the necessary switching [1].

The Amoeba architecture consists of four principal components, as is shown in Figure 1. First are the workstations, one per user, which run window management software, and on which users can carry out editing and other tasks that require fast interactive response [2]. Second are the pool processors, a group of CPUs that can be dynamically allocated as needed, used, and then returned to the pool. For example, the *make* command might need to do six compilations, so six processors could be taken out of the pool for the time necessary to do the compilation and then returned. Alternatively,

The authors were supported under a grant of the Netherlands Organization for Scientific Research (N.W.O. no. 125-30-10), and a grant of the Dutch Royal Academy of Sciences (KNAW) .

with a five-pass compiler, 5 x 6 = 30 processors could be allocated for the six compilations, gaining even more speedup [3].

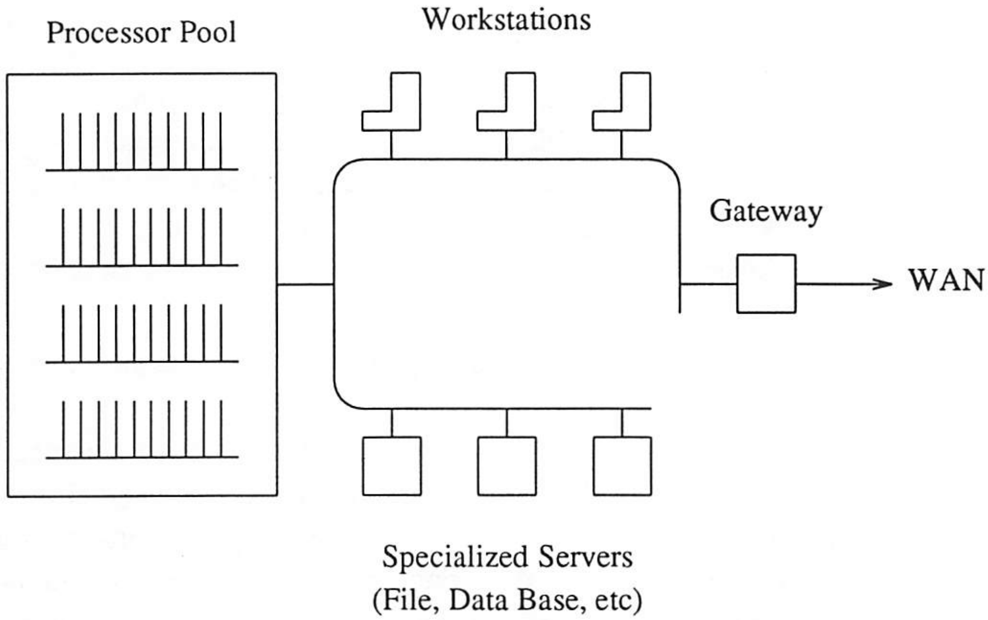


Fig. 1. The Amoeba System Architecture.

Third are the specialized servers, such as directory, file, and block servers, database servers, accounting servers, boot servers, and various other servers with specialized functions. Fourth are the wide-area network gateways, which are used to link Amoeba systems at different sites in possibly different countries into a single, uniform system. An Amoeba gateway works much the same way as a Mach NetMsgServer, except on a per site basis rather than per machine [4, 5].

All the Amoeba machines run the same microkernel, which primarily provides communication services and little else. The basic idea behind the kernel was to keep it small, not only to enhance its reliability, but also to allow as much of the operating system as possible to run as user processes, providing for flexibility and experimentation.

All objects in Amoeba are named and protected by *capabilities* [6, 7]. Capabilities, combined with remote procedure call, provide a uniform interface to all objects in the Amoeba system. A capability has 128 bits, and is composed of four fields, as shown in Figure 2:

1) The *server port*: a 48 bit sparse address identifying the server process that manages the object. A server can choose its own port.
2) The *object number*: an internal 24 bit identifier that the server uses to distinguish among its objects. The server port and the object number together uniquely identify an object.

3) The *rights field*: 8 bits telling which operations on the object are permitted by the holder of this capability.

4) The *check field*: a 48-bit number that protects the capability against forging and tampering.

48	24	8	48	# bits
Server Port	Object Number	Rights	Check Field	

Fig. 2. A capability.

When a server is asked to create an object, it picks an available slot in its internal tables, puts the information about the object in there along with a newly generated 48-bit random number. The index into the table is put into the object number field of the capability. The rights in the capability are protected by encrypting them together with the random number, and storing the result in the check field. A server can check a capability by performing the encryption operation again using the random number in the server's tables, and comparing the result with the check field in the capability.

Capabilities can be stored in directories that are managed by the *directory service*. A directory is effectively a set of <ASCII string, capability> pairs, and is itself just another object in the Amoeba system. Directory entries may, of course, contain capabilities for other directories, and thus an arbitrary naming graph can be built. The most common directory operation is to present an ASCII string and ask for the corresponding capability. Other operations are entering and deleting directory entries, and listing a directory.

2. The Amoeba Microkernel

Now we come to the structure of the Amoeba software. The software consists of two basic pieces: a microkernel, which runs on every processor, and a collection of servers that provide most of the traditional operating system functionality. In this section we will describe the microkernel. In the next one we will describe some servers.

The Amoeba microkernel runs on all machines in the system. It has four primary functions:

1. Manage processes and threads within these processes.
2. Provide low-level memory management support.
3. Support transparent communication between arbitrary threads.
4. Handle I/O.

Let us consider each of these in turn.

Like most operating systems, Amoeba supports the concept of a process. In addition, Amoeba also supports multiple threads of control, or just *threads* for short, within a single address space. A process with one thread is essentially the same as a process in UNIX.† Such a process has a single address space, a set of registers, a program counter, and a stack. Threads are illustrated in Figure 3.

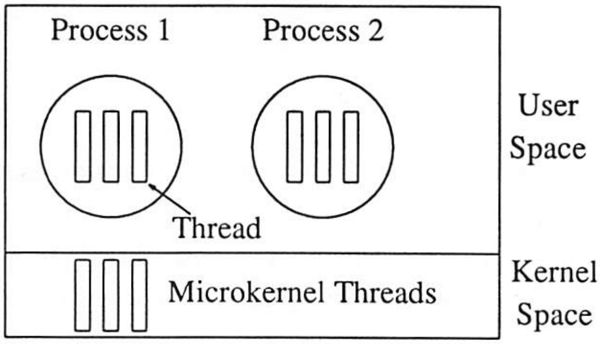


Fig. 3. Threads in Amoeba.

In contrast, although a process with multiple threads still has a single address space shared by all threads, each thread logically has its own registers, its own program counter, and its own stack. In effect, a collection of threads in a process is similar to a collection of independent processes in UNIX, with the one exception that they all share a single common address space.

A typical use for multiple threads might be in a file server, in which every incoming request is assigned to a separate thread to work on. That thread might begin processing the request, then block waiting for the disk, then continue work. By splitting the server up into multiple threads, each thread can be purely sequential, even if it has to block waiting for I/O. Nevertheless, all the threads can have access to a single shared software cache. Threads can synchronize using semaphores and mutexes to prevent two threads from accessing the shared cache simultaneously.

Threads are managed and scheduled by the microkernel, so they are not as lightweight as pure user-space threads would be. Nevertheless, thread switching is still reasonably fast. The primary argument for making the threads known to the kernel rather than being pure user concepts relates to our desire to have communication be synchronous (i.e., blocking). In a system in which the kernel knows nothing about threads, when a thread makes what is logically a blocking system call, the kernel must nevertheless re-

† UNIX is a Registered Trademark of AT&T in the USA and other countries.

turn control immediately to the caller, to give the user-space threads package the opportunity to suspend the calling thread and schedule a different thread. Thus a system call that is logically blocking must in fact return control to the "blocked" caller to let the threads package reschedule the CPU. Having the kernel be aware of threads eliminates the need for such awkward mechanisms.

The second task of the microkernel is to provide low-level memory management. Threads can allocate and deallocate blocks of memory, called *segments*. These segments can be read and written, and can be mapped into and out of the address space of the process to which the calling thread belongs. To provide maximum communication performance, all segments are memory resident.

The third job of the microkernel is to provide the ability for one thread to communicate transparently with another thread, regardless of the nature or location of the two threads. The model used here is a *remote procedure call* (RPC) between a client and a server [8]. Conceptually, the initiating thread, called the *client*, calls a library procedure that runs on the *server*. This mechanism is implemented as follows. The client, in fact, calls a local library procedure known as the *client stub* that collects the procedure parameters, builds a header and a buffer, and executes a kernel primitive to perform an RPC. At this point, the calling thread is blocked. The kernel then sends the header and buffer over the network to the destination machine, where it is received by the kernel there. The kernel then passes the header and buffer to a *server stub*, which has previously announced its willingness to receive messages addressed to it. The server stub then calls the server procedure, which does the work requested of it. The reply message follows the reverse route back to the client. When it arrives, the calling thread is given the reply message and is unblocked. The RPC mechanism is illustrated in Figure 4.

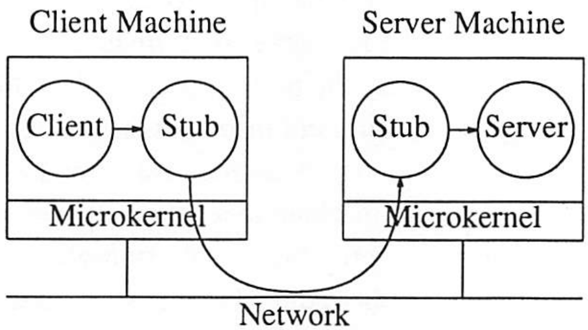


Fig. 4. Remote procedure call.

All RPCs are from one thread to another. User-to-user, user-to-kernel, and kernel-to-kernel communication all occur. (Kernel-to-user is technically legal, but, since that constitutes an upcall, they have been avoided except where that was not feasible. It is currently used for loading memory segments from the file server.) When a thread blocks

awaiting the reply, other threads in the same process that are not logically blocked may be scheduled and run.

In order for one thread to send something to another thread, the sender must know the receiver's address. Addressing is done by allowing any thread to choose a random 48-bit number called a *port*. All messages are addressed from a sending port to a destination port. A port is nothing more than a kind of logical thread address. There is no data structure and no storage associated with a port. It is similar to an IP address or an Ethernet address in that respect, except that it is not tied to any particular physical location.

When an RPC is executed, the sending kernel locates the destination port by broadcasting a special LOCATE message, to which the destination kernel responds. Once this cycle has been completed, the sender caches the port, to avoid subsequent broadcasts.

The RPC mechanism makes use of three principal kernel primitives:

- do_remote_op - send a message from client to server and wait for the reply
- get_request - indicates a server's willingness to listen on a port
- put_reply - done by a server when it has a reply to send

Using these primitives it is possible for a server to indicate which port it is listening to, and for clients and servers to communicate. The difference between *do_remote_op* and an RPC is that the former is just the message exchange in both directions, whereas the RPC also includes the parameter packing and unpacking in the stub procedures.

3. Performance of Amoeba RPC

All communication in Amoeba is based on RPC. If the RPC is slow, everything built on it will be slow too (e.g., the file server performance). For this reason, considerable effort has been spent to optimize the performance of the RPC between a client and server running as user processes on different machines, as this is the normal case in a distributed system. In Figure 5 we give our measured results for sending a zero-length message from a user-level client on one machine to a user-level server on a second machine, plus the sending and receiving of a zero-length reply from the server to the client. Thus it takes 1.1 msec from the time the client thread initiates the RPC until the time the reply arrives and the caller is unblocked. We have also measured the effective data rate from client to server, however, this time using large messages rather than zero-length ones. From the published literature, we have looked for the analogous figures for several other systems and included them for comparison purposes.

The RPC numbers for the other systems are taken from the following publications: Cedar [8], *x*-Kernel [9], Sprite [10], V [11], Topaz [12], and Mach [9]. The numbers shown here cannot be compared without knowing about the systems from which they

System	*Hardware*	*Null RPC in msec.*	*Throughput in kbytes/s*	*MIPS*	*Implementation Notes*
Amoeba	Sun 3/60	1.1	820	3.0	user-to-user
Cedar	Dorado	1.1	250	4.0	custom microcode
x-Kernel	Sun 3/75	1.7	860	2.0	kernel-to-kernel
V	Sun 3/75	2.5	546	2.0	user-to-user
Topaz	Firefly	2.7	587	5.0	5 VAX CPUs
Sprite	Sun 3/75	2.8	720	2.0	kernel-to-kernel
Mach	Sun 3/60	11.0	?	3.0	user-to-user

Fig. 5. Comparative Performance of RPC on Amoeba and other systems.

were taken, as the speed of the hardware on which the tests were made varies by about a factor of 3. On all distributed systems of this type running on fast LANs, the protocols are largely CPU bound. Running the system on a faster CPU (but the same network) definitely improves performance, although not linearly with CPU MIPS (Millions of Instructions Per Second) because at some point the network saturates (although none of the systems quoted here even come close to saturating it). As an example, in an earlier paper [13] we reported a null RPC time of 1.4 msec, but this was for Sun 3/50s. The current figure of 1.1 msec is for the faster Sun 3/60s.

In Figure 5 we have not corrected for machine speed, but we have at least made a rough estimate of the raw total computing power of each system, given in the fifth column of the table in MIPS. While we realize that this is only a crude measure at best, we see no other way to compensate for the fact that a system running on a 4 MIPS machine (Dorado) or on a 5 CPU multiprocessor (Firefly) has a significant advantage over slower workstations. As an aside, the Sun 3/60 is indeed faster than the Sun 3/75; this is not a misprint.

4. Implementation of RPC

A remote procedure call consists of more than just the request/reply exchange. The client has to place the capability, operation code, and parameters in the request buffer, and on receiving the reply it has to unpack the results. Moreover, it has to check the errors that might have occurred in the request/reply exchange. The server has to check the capability, extract the operation code, and parameters from the request and call the appropriate procedure. The result of the procedure has to be placed in the reply buffer. Placing parameters or results in a message buffer is called *marshalling*, and has a

non-trivial cost. We also have to handle different data representations in client and server. Also the capability checking might impose great overhead if not implemented carefully.

When a client invokes **do_remote_op** for the first time, a packet containing the server port is broadcast over the network to request the physical location of the server [14]. The kernel running the server responds with a packet containing its physical network address. The client caches this information so that it may use it as a hint in subsequent operations to the same server. Next the client sends the request packet, or a sequence of packets if the request does not fit in one packet, to the server using the acquired physical location. A retransmission timer is started to recover from network failures. Retransmissions are always sent to the same processor, since otherwise the at-most-once semantics cannot be guaranteed.

In the normal case in which a reply is generated quickly, the reply message is sent back and serves as the acknowledgement for the request. If the operation takes a long time, the client will retransmit the request. This time the server sends a separate acknowledgement. For a long remote operation special packets are exchanged to enquire about the status of the operation. Like requests, replies are split into several packets if they do not fit into one packet. Replies are separately acknowledged so that the server can start awaiting a new request immediately.

Special care needs to be taken to implement this protocol efficiently. First of all the coding has to be done carefully, since it turns out that the bottleneck in the communication is not the network, but in the processors that run the protocol. For example, unpacking densely packed messages are expensive operations. Second is the timer management. During a remote operation many timers need to be started, but they hardly ever expire, since they are canceled when an expected packet arrives. An efficient way of implementing the timers is using a *sweep* algorithm, that periodically checks whether the protocol is still progressing. If not, a message might be lost and a retransmission is in order.

Third is the context switching. Often when a thread blocks there are no other threads to schedule, since there are many processors available in Amoeba and the work is balanced over the different processors. In this case it is unnecessary to remove the thread from the run queue. When a packet comes in for this thread it can be restarted from where it stopped, and there is no overhead in putting it back on the run queue again. Also, when the message consists of several packets, the protocol management can be done at the interrupt level, and the thread does not need to be restarted at all.

RPC requests usually consist of a number of integer parameters and sometimes a request buffer consisting of bytes. Replies usually consist of an integer result and sometimes a reply buffer consisting of bytes. Since this is the common case we have optimized its implementation. For example, read and write operations on a file usually consist

of a buffer, an offset in the file, and a size. In the request and reply header we have reserved 8 bytes for parameters and results, which have been subdivided into two 2-byte words and a 4-byte word. These integer types have to be converted if the sender and receiver use different integer representations. The sender specifies which integer representation (little-endian or big-endian byte order) it uses.

More complicated data types can be handled by marshalling everything in the request and reply buffers. We leave the data representation in these buffers to the applications, but we have provided library routines that can be used to marshall common integer and floating point types in a machine-independent way. Work on a stub compiler is underway to have this done automatically.

The capability checking, if implemented naively, would involve expensive encryption for each operation. However, it is simple to cache the result of the encryption in the server, so that the encryption is hardly ever necessary. Cache entries are filled when capabilities are generated, or when the capability was not present in the cache. A simple least-recently-used algorithm guarantees a high hit-rate.

5. Summary

This paper presented a short description of the Amoeba distributed operating system. An Amoeba system consists of four types of components: dynamically allocatable pool processors, workstations, specialized servers, and gateways. Each component runs the same microkernel, but different types of components will typically run different user processes. The paper focussed in particular on the microkernel and the remote procedure call mechanism that it implements. We reported on the implementation and performance of RPC. Besides RPC, the Amoeba microkernel supports light-weight processes.

6. References

[1] F. Kaashoek, R. van Renesse, and H. van Staveren, ''The Fast Local Internet Protocol,'' *Conditionally accepted by ACM Trans. Comp. Syst.*, 1992.

[2] R. van Renesse, A. S. Tanenbaum, and G. J. Sharp, ''The Workstation: Computing Resource or Just a Terminal?,'' *Proc. of the Workshop on Workstation Operating Systems*, Cambridge, MA, November 1987.

[3] E. H. Baalbergen, ''Parallel and Distributed Compilations in Loosely-Coupled Systems: A Case Study,'' *Proc. Workshop on Large Grain Parallelism*, Providence, RI, October 1986.

[4] R. van Renesse, A. S. Tanenbaum, J. M. van Staveren, and J. Hall, ''Connecting RPC-Based Distributed Systems Using Wide-Area Networks,'' *Proc. of the 7th Int. Conf. on Distr. Computing Systems*, pp. 28-34, West Berlin, September 1987.

[5] R. van Renesse, J. M. van Staveren, J. Hall, M. Turnbull, A. A. Janssen, A. J. Jansen, S. J. Mullender, D. B. Holden, A. Bastable, T. Fallmyr, D. Johansen, K.

S. Mullender, and W. Zimmer, "MANDIS/Amoeba: A Widely Dispersed Object-Oriented Operating System," *Proc. of the EUTECO 88 Conf.*, pp. 823-831, ed. R. Speth, North-Holland, Vienna, Austria, April 1988.

[6] S. J. Mullender and A. S. Tanenbaum, "Protection and Resource Control in Distributed Operating Systems," *Computer Networks*, Vol. 8, No. 5-6, pp. 421-432, October 1984.

[7] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse, "Using Sparse Capabilities in a Distributed Operating System," *Proc. of the 6th Int. Conf. on Distr. Computing Systems*, pp. 558-563, Cambridge, MA, May 1986.

[8] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Comp. Syst.*, Vol. 2, No. 1, pp. 39-59, February 1984.

[9] L. Peterson, N. Hutchinson, S. O'Malley, and H. Rao, "The x-kernel: A Platform for Accessing Internet Resources," *IEEE Computer*, Vol. 23, pp. 23-33, May 1990.

[10] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch, "The Sprite Network Operating System," *IEEE Computer*, Vol. 21, pp. 23-26, February 1988.

[11] D. R. Cheriton, "The V Distributed System," *Comm. ACM*, March 1988.

[12] M. D. Schroeder and M. Burrows, "Performance of the Firefly RPC," *Proc. Twelfth ACM Symposium on Operating Systems Principles*, pp. 83-90, December 1989.

[13] R. van Renesse, J. M. van Staveren, and A. S. Tanenbaum, "The Performance of the Amoeba Distributed Operating System," *Software—Practice and Experience*, Vol. 19, No. 3, pp. 223-234, March 1989.

[14] S. J. Mullender and R. van Renesse, "A Secure High-Speed Transaction Protocol," *Proc. of the Cambridge EUUG Conf.*, Cambridge, UK, September 1984.

Microkernel Operating System Architecture and Mach[1]

David L. Black[2] *David B. Golub* *Daniel P. Julin* *Richard F. Rashid*
Richard P. Draves *Randall W. Dean* *Alessandro Forin* *Joseph Barrera*
Hideyuki Tokuda *Gerald Malan* *David Bohman*[3]

Abstract

Modular architectures based on a microkernel are suitable bases for the design and implementation of operating systems. Prototype systems employing microkernel architectures are achieving the levels of functionality and performance expected and required of commercial products. Researchers at Carnegie Mellon University, the Open Software Foundation, and other sites are investigating implementations of a number of operating systems (e.g., Unix[4], MS-DOS[5]) that use the Mach microkernel. This paper describes the Mach microkernel, its use to support implementations of other operating systems, and the status of these efforts.

1 Introduction

Microkernel architectures offer a new approach for operating system implementation. These architectures separate the portions of the operating system that control basic hardware resources (often called the operating system 'kernel') from the portions that determine the unique characteristics of an operating system environment, for example, a particular file system interface. In contrast, the traditional monolithic approach to operating system implementation spreads knowledge about the basic system structure throughout a single large kernel. By modularizing the implementation, a microkernel architecture offers improved support for constructing new system services, and configuring systems for specialized environments. These architectures can also simplify porting a system to a new hardware platform because almost all of the machine-dependent code is isolated in the microkernel. Finally, the use of common underlying services provides support for the coexistence and interoperability of multiple operating system environments on a single host. Experience with im-

---

[1]This paper is a revised version of a paper originally published in the *Journal of Information Processing* (Volume 14, Number 4).

[2]Dr. Black's address is: Research Institute, Open Software Foundation, 11 Cambridge Center, Cambridge, MA 02142. Dr. Rashid's address is: Microsoft Corporation, One Microsoft Way, Redmond, WA 98052. Mr. Bohman's address is: NeXT, Inc., 900 Chesapeake Drive, Redwood City, CA 94063. The address for the other authors is: School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

[3]This research was supported by the Defense Advanced Research Projects Agency (DOD) and monitored by the Space and Naval Warfare Systems Command under contract N00039-87-C-0251, ARPA Order No. 5993. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. government.

[4]Unix is a trademark of Unix Systems Laboratories, Inc.

[5]MS-DOS is a registered trademark of Microsoft Corporation.

plementations based on the Mach system demonstrates that these advantages can be achieved without sacrificing performance.

The Mach kernel [1] is an example of this modular layered approach to operating system design. Mach is a multiprocessor kernel that incorporates in one system a number of key facilities that allow the efficient implementation of those functions necessary to support binary compatibility with existing operating system environments. These mechanisms are intended not simply as extensions to normal operating system facilities but as a foundation upon which Unix and other operating system facilities can be built. We have built and continue to enhance implementations of Unix that achieve levels of performance and functionality competitive with commercial products.

This paper describes research implementations of operating systems based on the Mach microkernel. The results of this work range from mature prototypes that achieve high levels of functionality and performance to more experimental efforts that explore different areas of the design space. The next section describes the overall architecture of operating system implementations based on Mach. Sections 3 and 4 describe the basic Mach microkernel and some of the important concepts and components common to the various implementations. Sections 5, 6, and 7 describe the prototypes that we have constructed using three different implementation approaches, including their status, advantages, and disadvantages.

2 System Architecture

Mach implementations of other operating system environments are based on the observation that modern operating systems such as Unix and OS/2[6] are primarily distinguished by the programming environment they provide rather than the way they manage or manipulate hardware resources. It is both possible and rational to think of such systems not as operating system kernels but as application programs — servers or sets of servers that can provide client programs with specific programming abstractions.

These traditional operating system environments can be treated as a set of services implemented in terms of a more primitive set of system abstractions. This layered approach allows the most complex software layers to be built above a relatively simple system software kernel which can directly manage hardware resources in ways that can meet the needs of specialized environments. In particular, this approach implies:

Tailorability — Versions of traditional operating systems such as 4.3BSD, OS/2, and System V.4 can be treated simply as different applications which can be purchased separately and potentially run simultaneously with other OS environments. The Mach kernel also provides a base for the implementation of special purpose or customized operating system environments for applications such as network servers or databases.

Portability — Nearly all of the code which constitutes the operating system environment is independent of a machine's instruction set, architecture, and configuration.

Network accessibility — A complex operating system environment need not reside on the same machine as its clients. This allows specialized systems and embedded applications to take advantage of general operating system features without having to incorporate those features in their own system.

[6] OS/2 is a trademark of International Business Machines.

Extensibility — New operating system environments and versions can potentially be implemented or tested alongside existing versions.

Real-time — Traditional barriers to real-time support in Unix can be removed both because the kernel itself does not have to hold interrupt locks for long periods of time to accommodate Unix system services and because the Unix services themselves are preemptable.

Multiprocessor support — The underlying kernel does not need to support complex system functions (e.g., file systems or network protocols) which may be defined in ways which limit parallelism. Thus its functions can be more completely parallelized and its features tailored to parallel applications.

Multicomputer support — Efficient operating system support for multicomputers (multiprocessors whose processors do not share physical memory) is greatly simplified. Since the kernel only provides a small number of basic abstractions, it can optimize the mapping of each abstraction onto the distributed hardware. This mapping can be transparent to the implementation of operating system environments, because it does not change the functionality of the kernel abstractions.

Security — Many traditional operating system environments have defined features which are inherently insecure. A more minimal kernel can be defined and implemented in a secure fashion such that trusted computing environments can be implemented in terms of its secure abstractions. The modular architecture of such a layered system is better suited to trusted systems than the structure of traditional kernels.

Other such implementations have, however, frequently started from a rather different notion of the relationship between the system kernel and the supported OS environment: virtual machines (IBM's CP/67 [17]), layering the kernel on a simple message engine (AT&T's MERT [15]), using a global shared communication area (Taos [21]), or loading operating system environment specific emulation-assist code into the kernel (Chorus [2]). In contrast, our approach takes advantage of Mach's support for the manipulation of system resources via a small set of machine-independent abstractions and its integration of memory management and communication functions. All functionality pertaining to the implementation of specific operating system services is performed by Mach tasks (including the application itself) which take advantage of the Mach IPC, scheduling, and virtual memory services. No functionality specific to Unix or other emulated operating systems exists within the kernel.

3 The Mach Microkernel

Mach provides an unusually flexible execution environment for both system and user applications. It exposes the management of CPU, communication, virtual memory, and secondary storage resources in a way that allows system applications to use those resources efficiently.

3.1 Mach Kernel Features

The key features of Mach are:

- Task and thread management
- Interprocess communication
- Memory object management

- System call redirection
- Device support
- User multiprocessing support
- Multicomputer support

A more detailed description of Mach and its abstractions can be found in [1].

Task and thread management Mach supports the task and thread abstractions for managing execution. A task is a passive resource abstraction, consisting of an address space and communication access to system and server facilities. Computation within a task is performed by one or more threads; these threads share the address space and all other resources of the task. Threads are scheduled to processors by the Mach kernel, and may run in parallel on a multiprocessor. Multiple scheduling classes can be defined for threads. At the present time, two classes are provided: fixed priority and timesharing. Timesharing threads are scheduled by the Mach kernel using a multi-level feedback queue scheduler with 32 priority levels. The schedulability of tasks, their threads, and even processors can be controlled by user-state programs. In particular, it is possible for a privileged user-state program to directly control the mapping of threads to processors and thus fully determine system scheduling policy. This feature has been used to implement a number of multiprocessor scheduling policies including gang scheduling [4].

Interprocess communication Mach provides interprocess communication among threads via constructs called ports. Ports are protected by a capability mechanism so that only Mach tasks with appropriate send or receive capabilities can access a port. All services, resources, and facilities within the Mach kernel, as well as those exported by particular Mach tasks or servers are represented as ports. Mach tasks, threads, memory objects, and processors are, for example, all manipulated by sending messages to ports which represent them. As such, the Mach port facility can be thought of as an object reference mechanism. In addition, this facility can be transparently extended over a network by using external communications servers [19].

Memory object management The address space of a Mach task is represented as a collection of mappings from linear addresses to offsets within Mach memory objects. The primary role of the kernel in virtual memory management is to manage physical memory as a cache of the contents of memory objects. The kernel's representation for the backing storage of a memory object is a Mach port to which messages can be sent requesting or transmitting memory object data [25]. Memory object backing storage can thus be implemented by user-state programs such as file system servers, database applications or AI knowledge stores.

System call redirection The Mach kernel allows a designated set of system calls or traps to be handled by code running in user mode within the calling task. The set of emulated system calls needs to be set up only once; it is inherited by child tasks on fork operations. This feature allows the binary emulation of operating system environments such as Unix. It also allows for monitoring, debugging, and transparent extension of existing operating system functions. Similar facilities are provided for redirecting exceptions [5]; this is used to implement redirection for operating systems whose system call linkages are treated as exceptions by Mach (e.g., DOS).

Device support The Mach kernel provides low-level device support [10]. Each device is represented as a port to which messages can be sent to transfer data or control the device. Data is transferred through read and write operations; the request and reply messages are exported separately, allowing

both synchronous and asynchronous styles of I/O. The external memory object protocol allows a task to map the frame buffer for a graphics device directly into its address space.

User multiprocessing A user-level multithreading package, the C thread library [6], facilitates the use of multiple threads within an address space. It exports mutual exclusion mutex locks and condition variables for synchronization via condition_wait and condition_signal operations. This library has recently been improved to optimize the use of Mach kernel threads by multiplexing C threads onto kernel threads and implementing user-mode context switches between C threads that do not involve the kernel.

Multicomputer support Mach supports multicomputers (multiprocessors which use an interprocessor network instead of shared physical memory) by transparently mapping the Mach abstractions onto the distributed hardware [3]. For example, when a new task is to be created, Mach can create the task on any node in the multicomputer system. Kernel management of distributed hardware confers a number of advantages. Existing operating system environment implementations can be used without change because the multicomputer kernels support the standard Mach abstractions. The Mach kernel also serves as a single location for optimizations appropriate for multicomputers, allowing all uses of it to benefit from these optimizations. For example, when data is copied using Mach operations, Mach can use techniques such as copy-on-reference and copy-on-write to reduce the amount of physical memory which is actually copied. Finally, the location transparency of Mach abstractions simplifies the implementation of load balancing. A user mode resource manager can dynamically balance system load in a transparent fashion by migrating tasks among nodes. Mach's multicomputer support is applicable to conventional multicomputers (e.g., hypercubes and meshes) as well as processor pools formed by using a high performance network (e.g., FDDI) to connect workstations or shared memory multiprocessors.

3.2 Evolution from Mach 2.5

The Mach 3.0 microkernel has evolved from the Mach 2.5 system that is the basis for commercial systems from NeXT, Encore, OSF, Omron, and others. The Mach 2.5 system contains compatibility code for BSD Unix in the kernel and depends heavily on that code. For example, it is not possible to create a Mach task in a 2.5 system without also creating a Unix process. All of this compatibility code has been removed from the kernel in the Mach 3.0 system; this has resulted in the addition of Mach interfaces in areas that did not exist in 2.5 (e.g., devices). The major change to the Mach code has been a complete rewrite of the IPC implementation to achieve improvements in both memory usage and performance. By optimizing the representation of ports and port rights, the amount of memory used for IPC data structures was reduced by 50% for a system running a Unix emulation. New algorithms and other optimizations to favor the common remote procedure call case in both the IPC and scheduling code have doubled the speed of a null RPC [8][9]. Mach 3.0 now executes a null RPC on a DecStation[7] 3100 (16.67Mhz R2000 cpu) in 95 microseconds.

The use of continuations is an important contribution to the performance of Mach 3.0 [8]. As used by the Mach scheduler, a continuation is the address of a routine to call when a thread continues execution plus a small data structure that contains local state needed by that routine. This local state corresponds to the local variables that would normally be saved (e.g., in a control block structure). Saving and restoring this local state replaces the saving and restoring of processor registers during a context switch, representing a significant reduction in the quantity of data manipulated, especially for modern processor architectures with large numbers of registers. The use of a continuation also replaces the execution context (the routine that invoked the context switch and its call-

[7]DECstation is a trademark of Digital Equipment Corporation.

ers) that is normally saved on a kernel stack. Hence, a thread blocked with a continuation does not require a kernel stack, eliminating the need for many of these stacks and switches among them during scheduling operations. The result is a transformation of kernel stacks from a per thread resource into a per processor resource at a considerable savings in kernel memory usage. Continuations have been applied to the IPC, exception, and page fault handling facilities of Mach 3.0.

3.3 Real-Time Mach

The Mach kernel is similar in structure to real-time message passing kernels. It contains no built-in file system or other higher level facilities which could interfere with interrupt handling or real-time performance. As such, it provides a useful vehicle for experimental work in real-time scheduling and resource management. Due to the portability of the Mach kernel, Real-Time Mach should be able to provide a common real-time computing environment for various machine architectures including single board computers and embedded systems.

The Advanced Real-Time Technology (ART) group at Carnegie Mellon has developed real-time scheduling and resource management enhancements for the Mach kernel as well as a real-time application development toolset. This research effort is based on experiments carried out over a number of years on the ARTS distributed real-time kernel [24] and its real-time toolset, Scheduler 1-2-3 [22] and Advanced Real-Time Monitor [23].

The objective of the Real-Time Mach project has been to develop a real-time version of Mach that can support a predictable real-time computing environment and to develop an associated real-time toolset. Real-Time Mach supports the following real-time features: a real-time thread model, an integrated real-time thread scheduler including multiple policies, real-time synchronization mechanisms, and a memory resident memory object manager. Prototype Real-Time Mach kernels with these features have been implemented and are in use at Carnegie Mellon University.

4 Operating Systems as Application Programs

The basic facilities provided by the Mach kernel support the implementation of operating systems as Mach applications. The memory object management mechanisms allow paging functionality to be implemented outside the kernel, and provide system programmers with control over data cached by the virtual memory system. The system call redirection mechanism makes it possible to support applications that have the system call traps linked into their executable binaries without modifying the kernel. Hence, the structure of a user-mode operating system emulation consists of an emulation library for each application plus one or more servers. Operating systems that have been implemented in this fashion include 4.3BSD Unix, OSF/1[8], MS-DOS, and the Macintosh[9] operating system. Efforts to implement other operating systems (e.g., VMS) using this technology are in progress elsewhere.

4.1 Emulation Libraries

Emulation libraries are key architectural components that support the implementation of operating system environments. An emulation library functions both as a translator for system service requests and as a cache for their results. System service requests from an application in the environment are translated to requests for the Mach microkernel or other servers that are used to emulate the target environment. Results returned by these requests may be cached for future use. Emulation libraries are transparently loaded into otherwise unused portions of application address spaces; Mach's memory inheritance mechanism is used to implement inheritance of these libraries for child

[8] OSF/1 is a trademark of the Open Software Foundation.

[9] Macintosh is a trademark of Apple Computer, Inc.

tasks created by such applications. It is not strictly necessary for the emulation library to occupy part of the application's address space, but doing so optimizes data transfer between the library and application.

Mach's system call redirection mechanism is used to invoke emulation libraries from fully linked application binaries. These binaries request operating system services by executing hardware trap instructions (e.g., SVC); Mach's system call redirection facility forwards these trap invocations to the emulation library. This trap redirection is not necessary if the base library containing the system call stubs (e.g., libc) can be dynamically linked to the application at execution time. In this case, a new version of the base library can be substituted that translates invocations of these stubs directly into operations for the Mach kernel or other servers. Current emulation library code is always executed with its own stack instead of using the stack of the application making the system call. This allows emulation libraries to be used with applications which may be doing their own stack management or providing their own lightweight process mechanisms.

Emulation libraries are vulnerable to tampering by application programs because they reside in application address spaces and it is well within the capabilities of a user program to read, write, or remove that region of memory. As a result, care is taken to ensure that the correct functioning of the servers cannot be affected by malicious or unintentional tampering with emulation libraries. It is also important that information managed by the library not be more security sensitive than information otherwise available to the user. In this regard, the library must operate under restrictions similar to that of the standard C runtime library.

System call implementations using emulation libraries and system call redirection within a single address space do not suffer a performance disadvantage with respect to in-kernel system call implementations. These performance comparisons were made between Mach 2.5 and Mach 3.0 using an emulation library on a 25MHz HP-Vectra (80386 processor). The basic mechanism costs of calling and returning from a system call are identical for the in-kernel and emulation library cases (35 microseconds). This is to be expected because both mechanisms involve a single kernel entry and exit; when a system call is redirected to an emulation, the return to the caller is performed entirely in user mode without further kernel involvement. When the additional work of handling a simple system call (e.g., getpid) is added, the emulation library approach is slightly faster (64 microseconds) than the in-kernel approach (68 microseconds). This work represents the additional state that must be set up and torn down to create the environment for handling a Unix system call.

4.2 OS Environment Architectures

There are several possible alternatives for the structure of operating system environment implementations using Mach. The simplest approach is to implement all of the functionality in an emulation library. This is best suited to single user systems, as service requests from such systems are often easy to translate into requests for Mach or other servers. Examples of systems that have been emulated in this fashion include MS-DOS and the Macintosh operating system. Our work with this approach has made extensive use of the native operating system code for the original system.

The alternative approaches use one or more servers in addition to the emulation library. These server-based approaches are distinguished by the granularity of the implementation's decomposition into servers. At one end of the spectrum are large granularity decompositions that implement most of the emulated functionality in one server. Both 4.3BSD and OSF/1 have been emulated using this structure. At the other end of the spectrum are small granularity decompositions that employ a family of functionally specialized servers in addition to the emulation library. This approach

allows the reuse of servers among different operating system environments, and supports the coexistence and sharing of resources among such environments.

5 Native OS Systems

Native OS systems utilize an emulation library and the code of the original operating system to implement the system services used by applications. This approach has been used to emulate the DOS and Macintosh operating systems on top of Mach. In both cases, significant portions of the native operating system are allowed to run directly because many service requests for single user systems do not involve operations that require intervention by the Mach kernel. An example from the DOS implementation is that native DOS filesystem code is allowed to execute, because its invocations of low level BIOS functions (the only code that actually accesses the I/O device) are intercepted [18]. In the Macintosh system, the graphics primitives that write to the display are allowed to execute directly from the Mac ROMs once the display has been appropriately mapped.

The Mach exception mechanism is used to intercept and redirect service requests for both systems because their system call invocation mechanisms differ from Mach and Unix on the same hardware. The Mach kernel does not recognize these as system call invocations, and instead treats them as exceptions. In addition to system calls, both systems require interception of certain low level functions internal to the native OS. Intercepting BIOS invocations from DOS is straightforward because DOS uses a system call-like mechanism (software interrupt) to invoke BIOS operations[10]; these invocations generate Mach exceptions that are redirected to the emulation library. In the Macintosh implementation, certain routines in the Macintosh ROMs must be intercepted. This was done by taking advantage of a ROM patch facility. The patch facility supports replacement of ROM operations by RAM equivalents so that changes can be made if bugs should be found in the ROM routines. The Mach emulation uses this feature to replace certain ROM routines with emulation library implementations.

The emulation library plays a dual role in these systems. It is responsible not only for implementing system services (or invoking native OS code to do so), but also for virtualizing access to hardware devices. This is because many applications in these systems expect to access hardware devices (e.g., the display) directly without invoking a system service. As part of its support for multi-user and multi-application environments, the Mach kernel must protect and control access to devices. Virtual memory techniques are used to make devices accessible to applications. For example, an inaccessible region of memory can be placed in the region of address space where an application expects to access the control registers of a device. Attempts to access these registers cause exceptions, which allow the emulation library to perform the appropriate functions. In many cases (e.g., displays), the Mach device pager can be used to map the device buffer memory directly into applications at the expected location. An additional area of virtualization is the use of threads to emulate asynchronous device interactions. For example, emulation library threads are used to handle both disk and keyboard accesses.

Both the Macintosh and DOS systems expect to execute on a Mach kernel that is also emulating the Unix operating system. The respective emulators are initially loaded from a Unix file system, and Unix file services are transparently available to applications using both emulated systems (e.g., DOS uses the Network Redirector to access Unix files). In addition, the emulated systems can manage disk partitions using their private (non-Unix) on-disk filesystem layouts. The Macintosh system also supports the use of Unix and Mach applications under MultiFinder; a Macintosh application

[10] Not all BIOS invocations are intercepted; those that do not cause protection or device access conflicts are allowed to execute using the native BIOS code.

has been written that provides access to a Mach/Unix C shell. A similar application is being implemented to provide shell access to Windows 3.0 under DOS.

These system implementations are functionally complete, and support virtually all Macintosh and DOS applications without change and with similar performance. Among the DOS applications that we have used on top of Mach are business applications such as Lotus 1-2-3[11], WordPerfect[12], and Windows 3.0[13], games such as Wing Commander[14] and Space Quest IV[15], etc. The Macintosh system supports Multifinder, business applications (e.g., MacDraw 2.0[16], Excel[17], Powerpoint[18]), games (e.g., Beyond Dark Castle[19]), etc. Application performance is essentially indistinguishable from the native system for both systems; the use of the continuation RPC enhancements described in Section 3.2 has been a major contributor to this achievement. The use of virtual memory in the Mach kernel and these implementations imposes minimum hardware requirements on these systems. For DOS systems, an 80386 or 80486 processor is required, as other compatible Intel processors do not support virtual memory. Most display types (e.g., VGA, EGA) are supported, as are third party sound boards. The corresponding requirement for Macintosh systems is a Mac II, and sound is also supported.

6 Large Granularity Server Systems

Our implementations of large granularity server decomposition employ an emulation library that communicates with a multithreaded server specific to the operating system being emulated. This server, contained in a single task, is typically invoked via a Mach message exchange for each system call issued by application processes. In addition to managing system call emulation, the server may also act as an external memory manager for files and other data.

A large granularity operating system emulation of this kind is attractive for several reasons:

- The server is solely responsible for performing the emulation of all OS environment semantics. The structure of the server is, in fact, similar to that of an in-kernel implementation; it has global knowledge of all the information needed for the emulation. Internal context switching between threads can be extremely fast.
- The OS server is completely pageable and can make more efficient use of memory (by sharing data structures and stack space) than can a multiple server implementation.
- It can be relatively straightforward to transform an existing in-kernel OS implementation into such a server, because most of the code can be reused. This can make it easy to preserve both existing code and semantics. This could allow vendors with proprietary OS environments to more quickly take advantage of Mach as a basis for their systems.

The feasibility of this approach has been demonstrated by implementing a Unix server and associated emulation library for 4.3BSD Unix [11]. This single task Unix server works well and is in regular use. It currently runs on a variety of platforms, including DECStations (2100, 3100, and 5000),

[11] Lotus 1-2-3 is a registered trademark of Lotus Development Corporation.
[12] WordPerfect is a trademark of WordPerfect Corporation.
[13] Windows 3.0 is a trademark of Microsoft Corporation.
[14] Wing Commander is a trademark of Origin Systems.
[15] Space Quest IV is a trademark of Sierra On-Line.
[16] MacDraw is a trademark of Apple Computer, Inc.
[17] Excel is a trademark of Microsoft Corporation.
[18] Powerpoint is a trademark of Microsoft Corporation.
[19] Beyond Dark Castle is a trademark of Silicon Beach Software, Inc.

i386 PCs from multiple vendors, and Sun 3 machines. This system is functionally interchangeable with existing versions of 4.3 BSD/Mach on those machines. A similar system has been implemented to emulate OSF/1. The remainder of this section describes the implementation of these systems and how they take advantage of the features that Mach provides.

6.1 Unix Server

The bulk of Unix services are provided by the Unix Server. It is implemented as a Mach task with multiple threads of control managed by the Mach C threads package. Internal synchronization and process-switching within the Unix Server (e.g., sleep, wakeup, spl) are implemented by using the C threads package's mutex and condition variable functionality. A typical system configuration will have dozens of C threads allocated within the Unix Server. Most threads belong to a common pool which handle incoming requests from user processes. Several threads are dedicated to routines that, in a monolithic kernel, would be driven by hardware interrupts (device IO completion, timeout, network code). All communication with hardware devices is done through Mach's IPC facility. Figure 1 shows the organization of the Unix Server and its relationship to the Mach kernel.

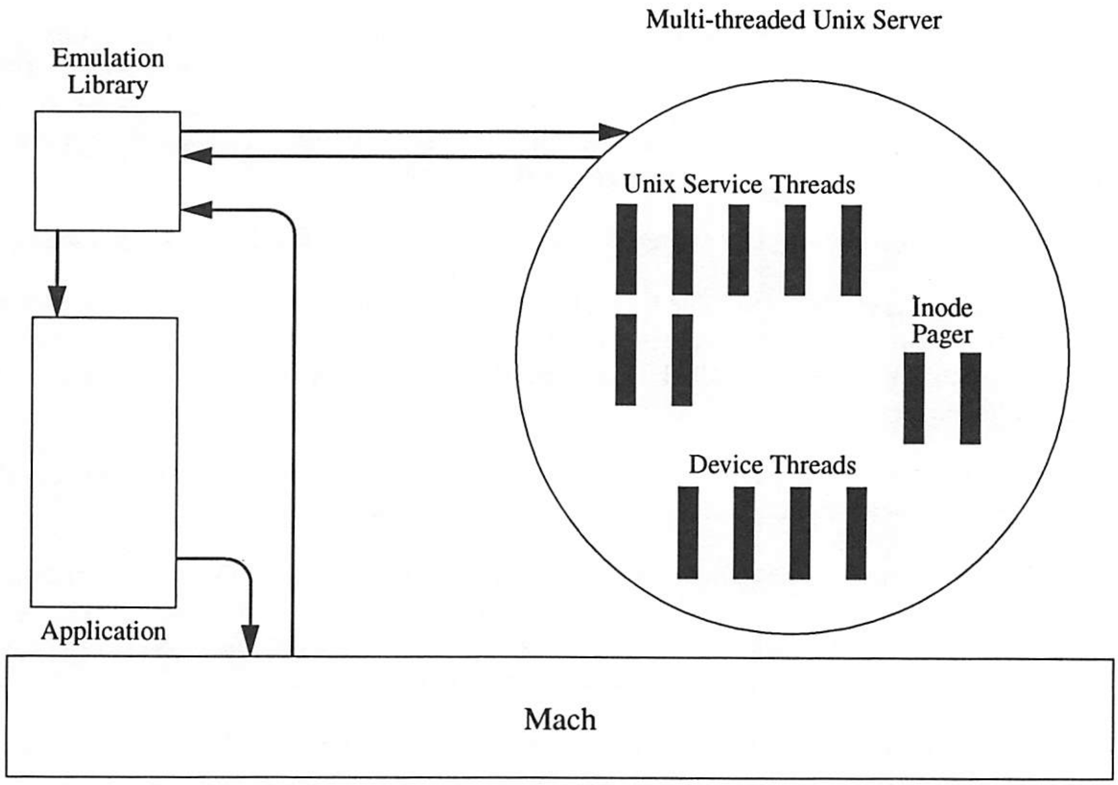


Figure 1: Unix Server System Architecture

The primary tool for communication between the Unix Server and a Unix application program is Mach IPC. Most requests for service arrive in the form of Mach messages requesting that a Unix operation or service be performed. For each incoming message, a C thread is dispatched from the pool to handle that operation. That thread then determines which Unix process requires service, what operation is to be performed and finally parses the message to obtain the arguments for that operation. Many, but not all, messages to the Unix Server correspond directly to system calls normally present in 4.3BSD.

The main departure from this style of interaction between the Unix Server and a Unix application can be found in the handling of 4.3BSD file access. Access to Unix files can be provided either through a pure message passing interface or through the Mach memory object facility. The decision of which interface to use can be made either by the Emulation Library or the Unix Server. The primary reason to choose a pure message passing interface would be performance in a network environment where a message passing interface corresponds more precisely to the natural implementation technology of a network. In a tightly coupled multiprocessor or a uniprocessor, a memory object interface is a more efficient way to transfer large amounts of data.

In the case of a memory object implementation of file access, the Unix Server acts as the memory object manager (or 'inode pager') for 4.3BSD files. When a file is opened by a Unix application its data is mapped directly into the portion of the Unix application address space occupied by the Transparent Emulation Library. That library then directly provides read, write, lseek, etc. system call access to the file's data. In order to ensure Unix file sharing semantics, the Emulation Library must hand-shake with the Unix Server through messages whenever conflicts with other applications could arise (see Figure 2). The major drawback of this approach is increased costs for open and close operations on files. These operations can require memory mapping and deallocation, which are reasonably expensive in practice. Our performance results have shown that open/close costs are important, but not nearly as important as the cost of read/write/lseek operations.

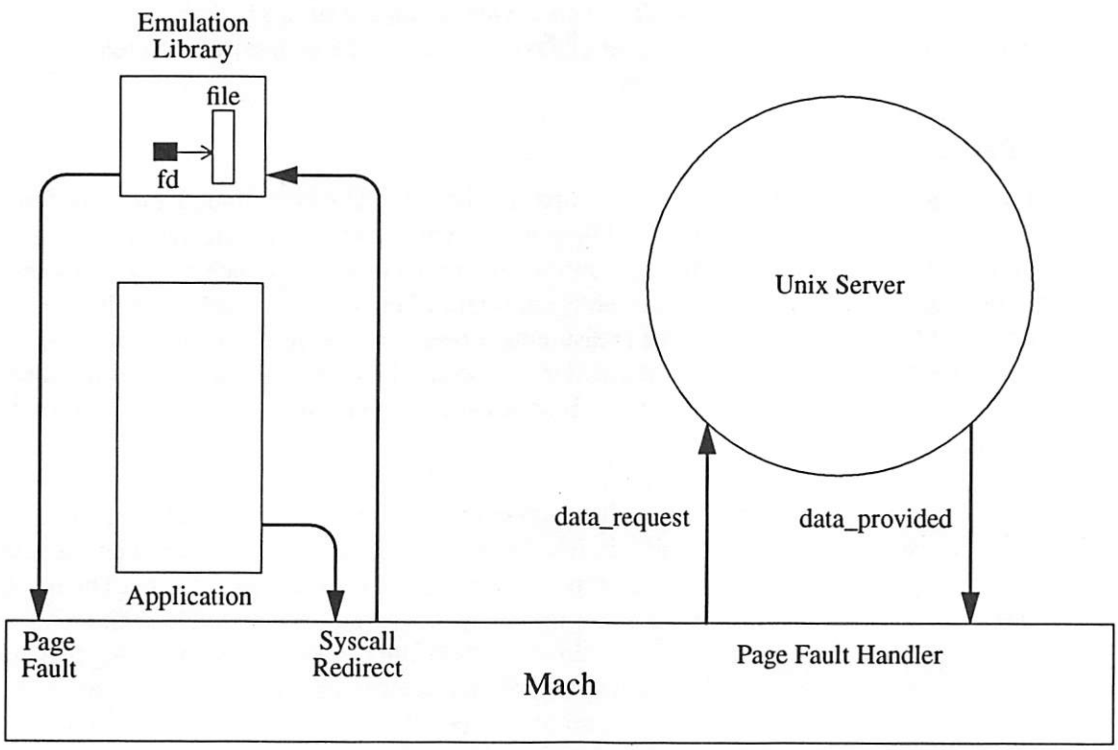


Figure 2: Unix Server Input/Output Architecture

6.2 Transparent Emulation Library

The Transparent System Call Emulation Library for this system contains the equivalent of the Unix system call handler and glue routines necessary to transform system calls into remote procedure

calls to the Unix Server. All application system calls are intercepted by the library using the Mach system call redirection facility. These calls may be directly handled by the library or transformed into remote procedure calls to the server. Most of the machine dependencies in the BSD code are handled by the Library. These include manipulating the application's stack for signal handling and forking a new process.

Many common Unix requests are handled exclusively by the Emulation Library. For example, a Unix file which has been mapped into the Emulation Library memory region as part of an open call can be directly read by the Emulation Library without requiring the intervention of the Unix Server. Shared memory techniques are used to allow most signal operations that set or read signal state (e.g., the signal mask) to be directly handled in the library. When the server forks a new process it creates two regions of memory shared between the server and the process, a read-only region for server to library communication and a read/write region for library to server communication[20]. The use of these techniques results in approximately one-half of application system calls being directly handled by the library without communicating with the server.

The Emulation Library is loaded into the address space of the initial user process. The Unix Server uses Mach's memory inheritance facility to cause inheritance of the Emulation Library by each child process from its parent on a fork operation. Server implementations of exec and similar operations which reload an address space with a new application program are careful to preserve the Emulation Library portion of the address space. One advantage of this technique is that it allows multiple Emulation Libraries with different behaviors (such as the support of different Unix variants) to coexist with the same Unix Server.

6.3 Performance

This section describes performance results comparing the 4.3BSD server running on Mach 3.0 with Mach 2.5 and commercial Unix systems. These measurements show that the system is achieving comparable performance to commercial systems even though its performance tuning and improvement are far from complete. We expect to obtain similar or better performance from the server implementation of OSF/1 after tuning. The measurement results come from several basic system call performance tests and two file system oriented tests, a compilation test and a more comprehensive file system test. The same disk with the same binaries and user environment was used for all direct comparisons.

The compilation test consists of a shell command file that runs nine compilations of small C source files. These files contained relatively few header file inclusions. Each compilation is separately timed using the 'time' command; the reported results are the total elapsed time. The resulting test stresses process creation/termination, program load and startup, file open/close and read/write costs for small files. The test performs approximately 2600 Unix system calls, including forking 57 processes, attempting to open 240 files and close 350 file descriptors, unlinking 100 files, and calling exec 160 times. Read, write, and lseek operations account for a large fraction of all system calls. Roughly 750 lseek operations, 450 read, and 230 write operations are performed.

The file system test was originally developed by M. Satyanarayanan for his performance evaluation of the Andrew File System [13]. Specifically, we used a version of the Andrew Benchmark modified by John Ousterhout [16]. This benchmark stresses directory and file creation, file copy,

[20] Use of this shared memory is optional; while it improves performance on uniprocessors and tightly coupled multiprocessors, it may not be appropriate for other architectures. If the memory is not configured, a message passing interface to the server is used instead.

file search (using 'find'), and compilation activity. A complete description of this benchmark can be found in the cited papers.

Table 1, Table 2, and Table 3 show some of the comparison results. The third column represents

Test	Mach 2.5	Mach 3.0	Mach 3.0 Speedup
create write (100K bytes) delete	634.1ms	596.0ms	1.06
lseek + read (8K bytes)	2.12ms	1.36ms	1.55
write (1M bytes)	0.26MB/sec	0.26MB/sec	1.00
read (cached)	4.40MB/sec	5.12MB/sec	1.16
read (uncached)	0.41MB/sec	0.38MB/sec	0.92
compilation	28.5 sec	27.4 sec	1.04
filesystem	400 sec	405 sec	0.98

Table 1: Mach 2.5 vs. Mach 3.0 + BSD Server: 8MB Sun 3/60 with Priam disk

Test	Mach 3.0	SunOS 4.1	Mach 3.0 Speedup
getpid	0.102ms	0.090ms	0.88
lseek + read (8K bytes)	1.36ms	1.56ms	1.14
read (cached)	5.12MB/sec	5.68MB/sec	0.90
compilation	26.1 sec	28.9 sec	1.10
filesystem	397 sec	373 sec	0.94

Table 2: Mach 3.0 + BSD Server vs. SunOS 4.1: 8MB Sun 3/60 with Wren V disk

Test	Mach 3.0	Ultrix 4.0	Mach 3.0 Speedup
create write (100K bytes) delete	281ms	436.72ms	1.55
write (10M bytes)	0.38MB/sec	0.43MB/sec	0.88
read (cached)	9.36MB/sec	5.82MB/sec	1.60
read (uncached)	1.01MB/sec	1.11MB/sec	0.90
compilation	11.4 sec	14.1 sec	1.23
filesystem	99 sec	100 sec	1.01

Table 3: Mach 3.0 + BSD Server vs. Ultrix 4.0: DECStation 5000/200 with Wren V disk

the relative speedup (or slowdown) of the Mach 3.0 plus server system against the other system. The variation in these numbers reflects the changed system architecture; moving the Unix implementation outside the kernel changes the relative speed of some operations with respect to each other. The important conclusion to draw from these results is that the overall performance of this system is comparable to Mach 2.5 and two commercial versions of Unix (and faster in some cases).

Work is continuing on measuring and tuning the performance of this and other areas of the system. Among our preliminary results in the networking area is a test in which ftp bandwidth is essentially unchanged when substituting a Mach 3.0 system for a Mach 2.5 system.

6.4 OSF/1 Unix Server.

Development of the large granularity emulation of OSF/1 (known as OSF/1 MK) has resulted in a number of improvements to this emulation technology. The OSF/1 system incorporates additional functionality not found in a BSD system, including STREAMS, a logical volume manager, and support for dynamic loading of shared libraries. The development environment has been enhanced to allow a Unix server to be run as a process under another Unix server. This second server approach permits use of a full featured debugger (e.g., gdb) on a server under development, and makes it possible to recover from the crash of such a server without a system reboot (only the second server crashes; the first server is still healthy). OSF/1's multiprocessor support improves throughput by removing bottlenecks to multiple threads inside the server (even on a uniprocessor); in contrast, the BSD server must restrict much of its code to one thread at a time. An unexpected benefit of the multiprocessor support is that it allows the removal of all interrupt protection logic (spls) from the OSF/1 server, resulting in a significant performance improvement. This is because multiprocessor code must use locks to protect against interrupts on other processors; in a server there are no interrupts, and these locks are sufficient to protect against interrupt code being executed by another thread. OSF/1 MK is functionally complete and has passed its validation tests, but performance tuning continues. A version of this system for NORMA-class multicomputers (no shared memory) is under development.

7 Small Granularity Server Systems

Our prototype small granularity server decomposition system divides the responsibility for operating system support among an emulation library and a collection of specialized servers (e.g., naming, authentication, file data access). The interfaces among the system components have a high degree of independence from target environments. The benefits of this independence include interchangeable components, code reuse, and portability. This section describes research work to design and build a small granularity system employing multiple servers for the emulation of multiple operating systems, and a prototype for Unix emulation [14].

The distinguishing feature of this system's structure is the use of a common object-oriented framework for server implementation. This framework is reusable, avoiding reimplementation for each new emulation system. This reuse extends to components of servers and even entire servers. The independence of this framework from the servers enhances system configurability, and makes it possible to support multiple concurrent system emulations on a single host.

7.1 Architectural Framework

Figure 3 shows the general organization of this system. Each process in the emulated system is implemented by a Mach task that contains the application and a copy of the emulation library that intercepts the application's system calls. Most of the target system's functionality is implemented by specialized servers such as a fileserver, network server, and process manager. Their services are exported to the emulation libraries via special libraries or proxies that facilitate and optimize client-server interactions. The final component is the Mach microkernel to provide the basic facilities for execution of the other components. This organization can be viewed as a combination of three independent software layers, namely the kernel, service (servers and proxies), and the rest of the emulation libraries.

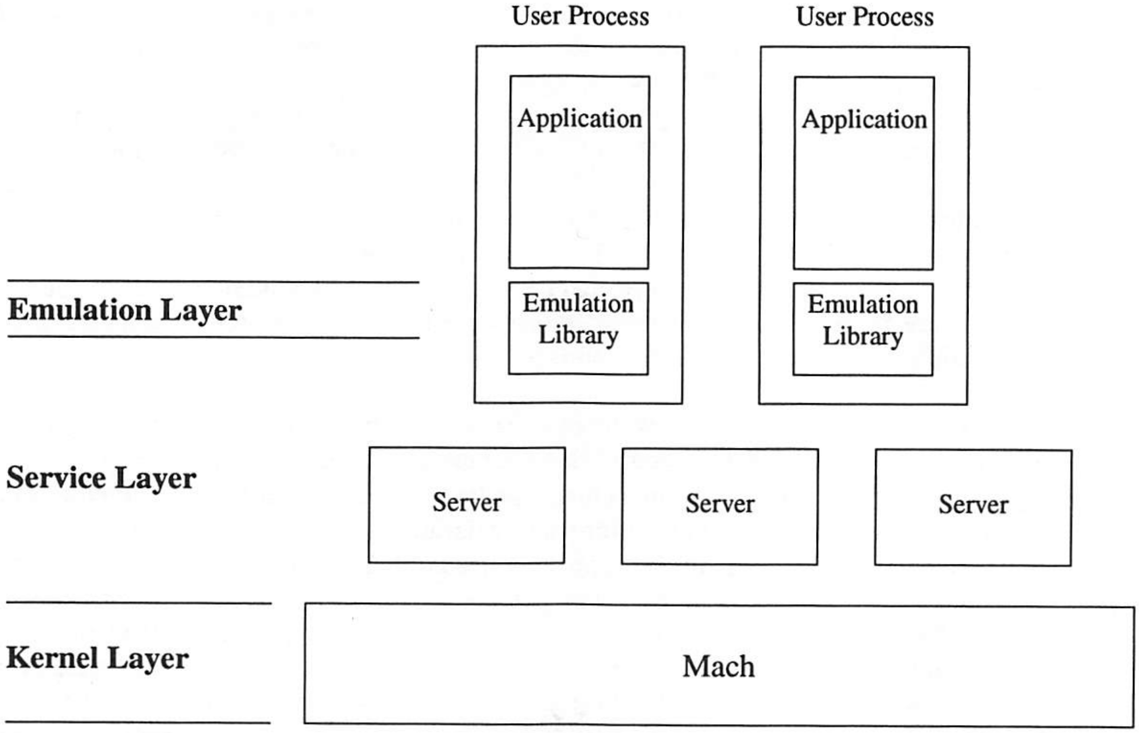


Figure 3: General System Structure

Although programming interfaces differ among operating systems, many of them provide similar sets of services in terms of functionality (e.g., file management, terminal I/O, network access, etc.). Hence the server layer can be structured as a set of components that can be assembled in various configurations. This enhances flexibility, modularity, security, etc. In many cases, the same components can be used for more than one implementation, based on the indirection provided by the emulation library. This decomposition of the service layer has two aspects:

- Decomposition or factorization of the functionality provided by the service layer into independent services. Each of these services is implemented in a separate module, usually as an autonomous system server. This decomposition isolates most problems and design decisions into independent, replaceable modules.

- Definition of standard interfaces and protocols for server-server and server-emulation library interactions. These generic interfaces do not map directly onto the set of services or servers, but correspond to groups of operations or mechanisms that may be common to several services. Such interfaces have been designed for both high level functions (e.g., access control and naming) and low level functions (e.g., locking and synchronization). These standard interfaces facilitate integration of servers into different system configurations. Many of the interfaces are relatively independent of the target operating systems; the emulation library is responsible for customizing them to specific target system interfaces. These interfaces are used only within the system, and are not exported to applications. Hence they form part of a System Programming Interface (SPI) including the Mach kernel interfaces (as opposed to an Application Programming Interface, or API).

To simplify the use and combination of servers, their interfaces are defined in terms of an object-oriented model. Each operating system service is represented by one or more items in this model. Examples from Unix include files, pipes, sockets, and ttys. These Unix abstractions may be represented by more general items, corresponding to the generic services explained previously. Servers typically implement a large number of similar, but independent items. Each item is opaque, and exports a well-defined set of operations. These operations can be invoked by the item's clients, usually the various emulation libraries operating on behalf of each application program. The server abstraction is not exported to clients; instead, clients have access to a number of items, and cannot determine which server manages any individual item. In addition to the interfaces, the various servers and items may also be implemented using object-oriented techniques; the word item is used to avoid confusion with implementation level objects.

An important aspect of the design is the optimization of client-server interactions by moving processing from system servers into clients. This optimization is based on the use of proxy objects [20]. A proxy consists of code loaded in a client's address space on behalf of the server; this code acts as a local representative to that client for a particular item from that server. Operations on the item are invoked locally by the client on the proxy instead of being directed to the server. A simple proxy may forward all invocations to the server, but more complex proxies may perform some processing or the entire operation locally, reducing the communication overhead and load on the server. Proxies are often used to cache information on the client side of a client-server interface. A new proxy object is instantiated in a client's address space as part of the protocol that establishes the client's access to the corresponding item. The set of all the proxies for the items currently accessed by a given client can be viewed as an additional layer of software within the service layer. Although they reside in the client's address space, proxies are logically part of the servers for their items; they are designed and implemented as part of the servers, rather than the clients

Another optimization of client-server interactions is based on the independence of server-proxy interfaces from both the item and target system interfaces. This allows responsibility for operation implementation (both item and target system) to be divided among the emulation library and one or more servers as needed. For example, many Unix process attributes, such as the file descriptor table and the signal mask, are managed by the emulation library without the use of any server or proxy.

7.2 Server Organization

There are two classes of servers in this system:

Application servers: provide services directly to application programs (e.g., a file server).

System servers: provide services primarily to other servers (e.g., nameservice, authentication).

The writer of an application server may assume the availability of the basic services provided by the system servers, and take full advantage of them. In addition, the framework provides standard low level mechanisms, interfaces, and reusable code fragments. These implement functions such as server invocation, access mediation and asynchronous event notification.

The current system contains the following application servers for Unix and other environments:

- One or more file servers supporting random access collections of bytes stored in various formats on local disks or remote file servers.
- A terminal server (TTY) for the management of logical and physical terminal devices.

- A local IPC server supporting basic communication between application programs (pipes, queues, Unix domain sockets, etc.).

- A process management server or task master to keep track of application processes and allow them to be operated on by external agents.

- One or more network servers providing access to the network and implementing various protocol families.

- A device server to control user access to the physical devices managed by the microkernel.

The current system supports only a Unix environment, but others are anticipated.

The application servers are supported by the following system servers, some of which are still under development:

- One or more root name servers, responsible for integrating the various servers together into a single uniform name space from which they can be located by clients.

- An authentication server, a secure repository for user identity information.

- A blackboard server to manage any information that must be efficiently shared among several emulated processes and/or servers.

- A lock/semaphore server to handle synchronization functions among clients and/or servers.

- A configuration/startup server to handle the startup of all the other servers and of the system as a whole, and to keep track of the system configuration (which servers to start, which devices to use, etc.).

- A remote Mach IPC server (netmsgserver) responsible for forwarding the Mach IPC facility over the network.

- A network shared memory server, providing uniform shared memory over a collection of nodes connected by a network.

- A diagnostics server responsible for logging all debugging, warning and error messages produced during the operation of the system.

This collection of application and system servers is sufficient to implement all of the functionality of Unix.

7.3 Status and Performance

The performance and functionality of the prototype for small granularity 4.3BSD Unix emulation has reached the point where the system is now self-hosting. While the performance is not comparable to either the large granularity emulation or commercial systems, it is sufficient for everyday use (and the system has the corresponding stability to support this). The major area of missing functionality is networking, as the absence of a TCP protocol implementation for this system prevents many Unix networking services from functioning; Unix sockets and the UDP protocol are implemented, allowing files to be retrieved and stored via tftp. Improving performance is a major area of our current work on this system.

A comparison with the large granularity server system provides some evidence that tuning this small granularity system will enable it to achieve competitive levels of performance. System calls in the large granularity system are handled either directly by the emulation library or by the library contacting exactly one server (the Unix Server). The same statement is true for this small granularity architecture; many system calls are handled in the library, and those that aren't almost always contact only one server. The performance results in Section 6.3 show that the large granularity system is achieving commercial levels of performance, so this operational similarity suggests that similar levels of performance should also be possible for this small granularity system.

Our current research also involves extending the functionality of this system. An important aspect is investigating the suitability of the overall framework for implementing operating systems other than Unix (e.g., OS/2). In addition to reusing the basic system servers, this work has a goal of reusing some application servers (e.g., fileservers) among different operating system implementations. Another aspect is the development of a trusted or secure system based on this technology. This effort targets the B3 level of security in the Trusted Computer System Evaluation Criteria [7], and is being pursued in cooperation with Trusted Information Systems, Inc. [12].

8 Conclusion

The Mach microkernel supports a variety of approaches to operating system implementation. Native OS systems allow most of the original system's code to execute without change; they are primarily applicable to single user operating systems such as DOS. Large granularity server systems concentrate most of the non-native operating system's functionality in a single server; this simplifies development by making it possible to reuse much of the original system's code. Small granularity server systems spread the implemented functionality across a collection of functionally specialized servers. The framework for constructing such systems can support reuse of components among different emulations, and interoperability of different emulations on a single host. A common feature of all of these systems is the use of an emulation library to intercept and implement system service requests made by applications, providing support for the execution of unchanged application binaries.

Microkernel architectures offer an important and viable implementation alternative to monolithic operating system architectures. Operating system implementations based on the Mach microkernel are achieving commercially competitive levels of functionality and performance. The maturity of these prototype implementations varies; some are in daily use (with acceptable performance), while others are still under active development. These architectures can support a variety of systems (including real-time and secure systems) on a common kernel base.

9 Acknowledgments

In addition to the authors, the developers of Mach and the operating system implementations discussed in this article include: Avadis Tevanian, Jr., Michael W. Young, Richard Sanzi, William Bolosky, Michael Jones, Jonathan Chew, Mark Stevenson, Douglas Orr, Robert Baron, Gregg Lebovitz, the ART group at Carnegie Mellon University and the research staff of the Open Software Foundation's Research Institute in both Cambridge, Massachusetts, and Grenoble, France.

10 Availability

Most of the software discussed in this paper is available as source distributions. No license or fee is required to obtain or use the Mach 3 microkernel. It can be obtained from Carnegie Mellon University via anonymous ftp (Name: anonymous, Password: <user>@<site>). Contact the host cs.cmu.edu (128.2.222.173), and obtain the file FTP.inst from the mach3 directory for further instructions. The large granularity emulation of 4.3BSD (single server) can also be obtained from

- **Ease of use** - Some operating systems can be difficult to install and operate on a computer, especially without additional instructions. DOS is not difficult to install and has a vast amount of publications and books available to assist users.

The **disadvantages** of DOS include:

- **Limited primary storage** - An applications program running in the DOS environment has maximum access to 640,000 bytes of primary memory. Many programs being written today, require access to more than 640,000 bytes.

- **"Single tasking" only** - Dos can only work with one user and a single applications program at a time. **Multitasking** is an operating systems ability to run several applications programs at the same time. The demand in the workplace for multitasking capabilities is growing rapidly.

- **Character-based interface** - Working with DOS, users issue commands or select menu items. This method requires more knowledge or experience with the application than a graphical user interface method.

The **Future of DOS:**

Industry observers believe that DOS will have a following of users for many years. However, because of its limitations and the greater power of other operating systems, DOS will lose its dominate role to other systems software programs.

DOS with WINDOWS (pages 41-43)

Windowing software extends the capability of DOS by creating an environment that allows **multitasking, dynamic data exchange, graphical user interface**, and **more primary storage.**

The **advantages** of enhancing DOS with Windows, include:

- **Multitasking** permits the computer to work on more than one application simultaneously.

- **Dynamic data exchange** enables one program to broadcast a request for information to other, currently running, programs.

- **Graphical user interface (GUI)** permits the use of a mouse to point at icons or pull-down windows for the purpose of performing tasks previously requiring the user to type a command.

- **More primary memory** can be managed by the windows program. DOS can only access 640 kb of primary storage by itself. DOS with Windows can access billions of bytes in primary storage.

The **disadvantages** of Windows, include:

- **Technological limitations** are still inherent because Windows is based on old technology (DOS). While windows does enhance the use of DOS there are still application limitations.

- **Limited applications** will be available to meet the needs of the future. This is based on the needs for programs that exceed the capabilities of DOS and Windows.

- **Network capabilities** of Windows applications are limited because this program was not originally designed to work with networks. Thus, it is not very efficient for the network environment.

OS/2 (pages 43-45) ***TM 3.04***

OS/2 (Operating System/2) has been jointly developed by IBM and Microsoft for IBM's Personal System/2 (PS/2) line of computers. It has the promise of being much more flexible than MS-DOS and will give the user **multitasking** capabilities (running several programs simultaneously).

Versions of OS/2 include:

Standard Edition 1.2 - IBM's first version runs on IBM PC AT, PC-XT Model 286, and PS/2 Models 50,60,70, and 80. This system does multitasking.

Standard Edition 1.1 - Is an upgrade of 1.0. It includes Presentation Manager (the graphical user interface).

Extended Edition - Is not an upgrade, but a separately sold product that has a built-in database and communications manager.

Advantages of OS/2 include:

- **Common User Interface** - applications written in OS/2 have a common graphics interface that is consistent when working with micro, mini, and mainframe computers.

- **Multitasking** - OS/2 has true multitasking capabilities that allows more than one application program to share the CPU. It is not constrained by the older DOS operating system as is the case with Windows. It is designed to maximize the performance of the newest mast powerful microcomputers.

- **Networking** - OS/2 is designed for networking applications meaning it can share data and programs among several computers.

The **Disadvantages** of OS/2 include:

- **Expense** - The start up cost for OS/2 is expensive. Start up cost can be nearly 3 times greater than a DOS with Windows system.

- **Fewer applications** - Applications software developers state it is difficult to write programs for OS/2. Since OS/2 is relatively new, there are not that many users of it yet. This narrow market share makes the price of programs for OS/2 expensive. However, as OS/2 develops more of a following, the number of applications will grow and the price of them will drop. However, many programmers have opted to write programs for windows, which is presently in greater demand, rather than OS/2.

MACINTOSH OPERATING SYSTEMS (pages 44-46) ***TM 3.05***

The "graphical user interface" was available in the Apple Macintosh even before OS/2. The Macintosh has had a successful following but has not really had wide spread consideration as a business computer system.

Versions

The operating system of the Macintosh is not as clearly defined as other microcomputer operating systems. Early in the Macintosh's development, a purchaser received System ad Finder diskettes. These two worked together to act as a standard operating system. Later, Apple introduced new models of the Macintosh computer. To support this new hardware, Apple changed the operating system and new versions of the System and Finder files were introduced.

Apple introduced new models on frequent intervals and there are several unique versions of its operating system. Unfortunately, these versions are not "backward compatible" (not like DOS), meaning that many applications designed for the newer Macintosh will not run on the older versions of the Macintosh.

Advantages of the Apple Macintosh include:

- **Quality graphics** - Macintosh has established a high standard for graphics. This has been the principal reason for its success because users can easily merge pictures and text to produce near professional quality output.
- **Ease of use** - The graphical interface is very popular with novices to the microcomputer. The reason for the popularity is because the need and effort required for learning to use this operating system is half as much as that of learning DOS.
- **Consistent graphics interface** - The user has similar screen displays, menus, and operations across all applications.
- **Multitasking** - The Macintosh System 7.0 has multitasking capability sharing the same CPU.
- **Communications between programs** - The System 7.0 allows programs to easily share data and commands with other applications programs.

Disadvantages of the Apple Macintosh include:

- **Lack of serious consideration as a business machine** - The general perception of the business community has not been favorable to the Apple Macintosh as a business computer.
- **Difficulties in compatibility** - In the past, DOS and Macintosh have had incompatible microprocessors. This has been a major stumbling block for Apple because businesses desire the compatibility and connectivity with the DOS world. Hardware and software are now available for the Mac that allows it to run DOS applications. Recently, Apple has developed links with Digital Equipment Corporation (DEC). They have made DOS-compatible circuit boards available and have developed ways of connecting the Macintosh into communications networks that use DOS.

UNIX: The "Portable" Operating System (pages 47-52) *TM 3.06*

Originally, **UNIX** was developed by AT&T as a minicomputer operating system; it now runs on microcomputers. Its popularity exists because for many years it has been used at universities. As the computer science graduates became employed, many of them promoted it as the operating system to use to their respective employers.

This operating system is less well known to the business community.

The recent development of the '386 chip technology has created a situation where UNIX has the possibility of competing with OS/2. The reason for this is because OS/2 is new and untested, however, UNIX is available right now.

Versions

The principal microcomputer versions of UNIX include: AT&T's UNIX System V; the University of California's Berkeley 4.2 UNIX; Sun Microsystems's SunOS; Microsoft has written its version called Xenix; Microsoft and AT&T are attempting to merge their versions to provide one standard; the Open Software Foundation has UNIX-OSF; IBM has AIX; APPLE has A/UX; and Apollo Computer has Domain/IX UNIX.

Advantages of UNIX include:

- **Portable programming language** - This means that it is able to be used with many different computer systems. The other three operating systems are not nearly as portable as UNIX.

- **Multitasking** - Like OS/2, UNIX has multitasking capability. This means that the CPU can be shared to do more than one program at a time.

- **Multiuser** - UNIX can also share the CPU and the system resources with multiple users. OS/2 does not have this capability, however, because hardware costs have dropped substantially, this is not as big a factor in the selection of an operating system.

- **Not limited by primary storage** - UNIX does not have the hardware restrictions like those of DOS and OS/2. It can do many operations that were previously only available in the minicomputer environment or larger. This is a significant benefit because a company can achieve the same type of performance with a microcomputer that previously required a larger computer system.

- **Networking** - Unix is able to share files over electronic networks with many different kinds of computer equipment. OS/2 Extended Edition promises this same service, but UNIX has successfully been accomplishing this for several years.

Disadvantages of using UNIX include:

- **Limited applications software** - There are many engineering applications programs, but there are very few business applications programs. Businesses are very dependent on running off-the-shelf software. The few applications that do exist usually require customizing, usually an expensive proposition. Many DOS users lack the experience to customize UNIX programs.

- **No UNIX standard** - There is no UNIX standard at any level. As discussed above, there are several different versions of UNIX available. The significance of this is when an application is written for one version of UNIX, it will not likely work in a different version of UNIX. An organization called the Open Software Foundation is trying to create a standard UNIX.

- **No graphical user interface** - Like with the UNIX standard, there is no standard graphical interface; there are several available.

- **Difficult for the novice** -The commands in UNIX tend to be cryptic and are difficult for new users to learn.

DOS in TRANSITION (page 48) ***TM 3.07***

What will replace DOS?

What is the best choice for an operating system?

These questions are difficult to answer even for those considered to be "in-the-know" regarding computers. Although other operating systems may be easier to use, DOS has a great many users. And, at present, other operating systems do not seem to offer many of these users enough benefit to make them want to change operating systems.

As we prepare for the future, some observers believe **windowing software** and Macintosh will be the solution for many users who are waiting for the dust to settle to determine the best choice of an operating system for the future, especially home users and small businesses.

Large organizations, universities, and major governmental units will require the power of OS/2 or possibly UNIX.

REVIEW QUESTIONS

1. What, in a phrase, is the difference between applications software and systems software?

   Application software programs are computer programs that can perform useful work. Systems software are programs that help the computer to manage its own resources. Part of the system software includes the operating system, which makes it easier for you to use the hardware.

2. What are four reasons for learning about systems software?

   To make an informed decision when selecting a microcomputer operating system, an end user should learn about the importance and purpose of systems software. This knowledge should include:

   1. knowing the predominate microcomputer system software packages that are presently available.
   2. knowing the limitations and strengths of each of the system software packages.
   3. knowing what capabilities are desired so that the hardware can be used to its full potential.
   4. the possibility of having to work with more than one system software environment within an organization.

3. Describe what a bootstrap loader does.

   The bootstrap loader is stored permanently in the computer's electronic circuitry to perform two essential tasks:

   1. it starts up the computer when you turn it on.
   2. it obtains the operating system from the default disk and loads the operating system into primary memory.

4. What do diagnostic routines do?

Diagnostic routines are programs stored in the computer's electronic circuitry. These routines automatically start up when the computer is turned on. They test the primary memory, the central processing unit, screen, keyboard and other components of the computer system. These routines are for the purpose of determining if the computer system is running correctly.

5. What is the basic input output system?

The basis input output system consists of service programs that are stored in primary memory. These programs enable the computer to interpret keyboard strokes and/or to transmit characters to the monitor or to a disk.

6. What are utility programs?

The operating system contains one set of programs called utility programs. These programs perform routine tasks such as formatting (or initializing) blank disks. This set of programs are known as external programs because they reside on the disk after DOS is loaded.

7. What is another name for initializing a blank disk?

Formatting.

8. What is an "IBM-compatible" microcomputer?

An IBM-compatible microcomputer is designed to operate using the same system software that was developed for IBM by Microsoft Corporation. These IBM-compatible computer systems are often called clones. Meaning a system that can use the same operating system (or system software) as is used in the IBM (or MS-DOS system software).

9. Name two principal limitations of DOS.

Two limitations of DOS are limited primary storage (640K) and its single tasking capability.

10. What are the principal differences between DOS and DOS with Windows?

Windows has multitasking, dynamic data exchange, graphical user interface, and can utilize more primary storage. DOS, while it is the most popular operating system in existence, is limited by memory, single tasking, and a character-based interface.

11. What is meant by multitasking?

Multitasking is the capability of more than one application program to share the CPU at the same time.

12. What is a graphical user interface?

A graphical user interface is a graphical windowing system. For every program running a window appears on the screen. These windows can be repositioned, made larger, smaller, expanded to fill the whole screen, and even shrunk to the size of a single character.

13. What is a microprocessor? Will OS/2 run on Intel 80286 and 80386 microprocessors?

A microprocessor is a tiny microchip that contains the CPU. It performs the manipulations on the data that is input into the system based on the instructions it is processing from programs.

Yes, OS/2 was designed to specifically run on the 80286 and 80386 chip technology.

14. Give three advantages of OS/2 over DOS.

OS/2 has several advantages over DOS. They include the protected mode that separates programs within memory, multitasking capability, access to greater amounts of primary memory, the graphical user interface, and communications between programs so they can share data and information.

15. What are two advantages of the Macintosh system software?

The primary advantages of the Macintosh are its quality graphics that can create near professional quality output with text and pictures and its ease of use.

16. What are two disadvantages of the Macintosh when used in a business environment?

The Macintosh has problems with compatibility with other computer systems and it is a relatively expensive system because of the lack of direct competition of compatible systems.

17. UNIX is said to be portable. What does this mean?

UNIX can be used with many different computer systems. As long as you are using the same version of UNIX in the other computer system. Thus, an application that runs on one system using UNIX will run on a different system using the same version of UNIX.

18. What is meant by the term multiuser?

A multiuser system means that the CPU is shared with several programs and with several different users all at one time.

19. What are four advantages of UNIX?

UNIX has the capability of running in a multiuser environment; it is not limited by primary storage; it permits multitasking; and, it is able to share files in a network environment.

20. What are three disadvantages of UNIX?

UNIX has limited applications software available; it lacks a standard at any level; and, it lacks a standard graphical interface.

CHAPTER 4
THE CENTRAL PROCESSING UNIT
Chapter at a Glance

I. The Four Types of Computer Systems

- A. Microcomputer
  1. word size
  2. storage capacity
  3. clock speed
  4. cost
  5. workstation
  6. portable
     - a. laptop
     - b. transportable

- B. Minicomputer
  1. word size
  2. storage capacity
  3. clock speed
  4. cost

- C. Mainframe computer
  1. word size
  2. storage capacity
  3. clock speed
  4. cost

- D. Supercomputer
  1. word size
  2. storage capacity
  3. clock speed
  4. cost

II. The CPU

- A. Control unit
- B. Arithmetic logic unit

III. Primary Storage

- A. Define its purpose
- B. Describe how it works

IV. The Binary System

- A. Identify its practical use with computers
- B. Describe the classifications in bytes (characters)
- C. Translate the two popular coding schemes
  1. ASCII
  2. EBCDIC
- D. Parity Bit - Error Checking

V. The System Unit

A. system board
B. microprocessor chips
C. memory
   1. random access memory (RAM)
   2. read only memory (ROM)
D. system clock
E. expansion slots
   1. open and closed architecture
   2. expanded memory
   3. display adapter
   4. additional storage
   5. multifunction boards
F. bus line architecture
   1. ISA - Industry Standard Architecture
   2. MCA - Micro Channel Architecture
   3. EISA - Extended Industry Standard Architecture
G. ports
   1. serial
   2. parallel
   3. peripheral devices

OBJECTIVES

The student should be able to:

1. identify the four classifications of computers.
2. describe the two main parts of the central processing unit and identify their function.
3. define main memory and the units used to measure it.
4. understand and describe how a computer uses binary code to represent data in electrical form.
5. identify the components of the system unit in a microcomputer.

VOCABULARY

address
ALU
ASCII
binary system
bit
bus line
byte
closed architecture
CPU
control unit
display adapter
distributed data processing
EBCDIC
EISA
EPROM
expanded boards
expanded memory
expansion slots
gigabyte
ISA
kilobyte
laptop portable
main memory
mainframe
MCA
megabyte
megahertz
microcomputer
millisecond
minicomputer
microprocessor chip
mother board
multifunction
neural networks
open architecture
optical computing
parallel port
parity bit
peripheral device
personal computer
processing cycle
PROM
RAM
RISC
ROM
register
serial port
supercomputer
superconductors
system board
system clock
system unit
terabyte
transportable
word size
workstation

THE FOUR TYPES of COMPUTER SYSTEMS (pages 54-60) *TM 4.01*

The four classifications of computer systems are microcomputers, minicomputers, mainframe, and supercomputers. These classifications have become less distinct over time, but are generally categorized by clock speed, amount of memory, storage capacity, costs, and word size (ie, 8, 16, 32, 64-bit processor).

Microcomputers are probably the most familiar kind of computer to the average individual. Microcomputers are often called **personal computers** because they can usually perform most of our personal computing needs at home and in the office. Their are sub-classifications of the microcomputer that include: laptop, transportable, and supermicros. COSTS: $200 - $15,000.

Workstations - The concept of workstations is changing dramatically due to technological advances. Until recently, workstations were expensive, powerful machines that engineers, scientists and other used for multitasking and powerful networking links. The powerful processor of the work station was the major difference between a workstation and a microcomputer. Today, the microprocessors used in many popular personal computer systems are providing workstation capabilities to end users, like us.

Transportable computers weigh from 14 to 30 pounds. They have greater computing power and better resolution than what is typical of a lightweight. Some transportables are not battery powered and must have a conventional power source.

Portable computers are also growing in popularity because of the more powerful and smaller size of this class of computer system. Lightweight portables ranging from just under 5 to 14 pounds are called **laptops**. They typically run using batteries and can fit into a brief case or be carried using a shoulder strap. Some small form of these portables are called **phonebook**-size (weigh between 8 and 10 pounds) and **notebook**-size (weigh less than 8 pounds) laptops.

Sublaptops are hand-held or pocket-size portable computers that weigh as little as one pound and are too small to fit in one's lap. These **palmtops** are intended to complement personal computers, not replace them. They can connect with desktop computers or networks and exchange data with them.

Minicomputers were developed as special-purpose mainframe computers. Generally, they are faster and can store more than a microcomputer and are slower and store less than a mainframe. However, there are exceptions that make it difficult to draw a clear line of distinction between the classifications.

Mainframe computers can process millions of instructions per second. They are often used as a decentralized computer system meeting the computing needs of banks, airlines, insurance companies, and most of the large corporations.

Supercomputers can process over one billion program instructions per second that are measured in **nanoseconds** (billionths of a second) and possibly even in **picoseconds** (trillionths of a second). A select few have the computing needs to justify the expense of a supercomputer system. It is used principally for worldwide weather forecasting, oil exploration, and weapons research.

The CPU (pages 60-61)

The **Central Processing Unit** is the part of the computer system that executes the program instructions. It consists of the **control unit** and the **arithmetic-logic unit**.

The **control-unit** is the part of the CPU that tells the computer system how to carry out a program's instructions. It directs electronic signals between main-memory and the arithmetic-logic unit. The control unit also controls signals to and from the input and output devices. The **arithmetic-logic unit** (ALU) performs arithmetic operations (+,-,/,and *) and logical operations (=, <, and >) which compare two pieces of data.

PRIMARY STORAGE (pages 61-63) ***TM 4.02***

The **main memory** (also known as primary storage, internal storage, RAM, and memory) is the part of the CPU that holds data for processing, the program (instructions for processing the data), and information (the processed data) that is waiting to be output or stored in secondary storage. The size of main memory can vary greatly from one computer system to the next. Some microcomputers ten years ago had as little as 16,000 characters of data storage capacity. Today, many microcomputers users would say that 640,000 characters of data storage capacity is necessary to run their application programs. However, technology and software have developed which allows the 640K threshold to be broken. An IBM Personal System/2 Model 80 can hold up to 16 million bytes.

In the near future, 640K will be inadequate for many of the newer applications programs that are being developed.

Registers are additional storage locations which are part of the control unit and ALU. They make processing more efficient because they act as a high speed staging area that holds data and instructions temporarily during processing. The contents held in registers can be handled much faster than the contents held in primary memory.

The processing cycle involves the interaction of primary storage and the CPU to process information. The computer stores characters of data or instructions in main memory at locations known as **addresses**. During each cycle of the computer's clock, a data word is fetched, decoded, executed and stored. The main memory uses an **address** that keeps track of the location of stored data and instructions. The address is a permanent label while the contents stored at a particular address can change continuously. Keep in mind that main memory is a temporary storage that will hold data/instructions only while the computer remains on or until another piece of data or instruction takes its place.

THE BINARY SYSTEM (pages 63-65) ***TM 4.03***

A microcomputer is made up of electronic circuitry that controls electronic signals that represent data and instructions. These electronic signals have two conditions--ON and OFF. When a program or a user communicates with the computer it must all be translated into ON and Off signals.

A convenient number system to use when working on the computers level is the **binary number system.** The binary system has only two digits--0 and 1. In the computer, the 0 represents an "off" signal and the 1 represents an "on" signal. Each 0 and 1 is called a **bit** (**bi**nary digi**t**) in the binary system. In order to represent numbers, letters, and special characters, bits are combined into groups of eight bits called **bytes.** A **byte** is the equivalent of one character, which in many computer systems is one addressable storage location. The capacity of memory and secondary storage is expressed in numbers of bytes:

A **kilobyte (K, KB,** or **K-byte)** represents 1,024 bytes (two to the power of ten).

A **megabyte (MB,** or **M-byte)** represents 1,024,000 bytes.

A **gigabyte (GB, or G-byte)** 1,024,000,000 bytes and is used as a unit of measure for only the largest mainframe computers or supercomputers.

There are two popular coding schemes that are used in the conversion of a character (that we understand) into a byte of 0s and 1s (that the computer understands).

Extended binary coded decimal interchange code (EBCDIC pronounced "eb-see-dick") is used predominately by mini- and mainframe computers and **American standard code for information interchange (ASCII** pronounced "as-key") is most widely used as the binary coding scheme for microcomputers. When a key is pressed on a computer keyboard, it is automatically converted into the binary form that the computer recognizes.

The **parity bit** is often employed as a means of having the computer detect if it is receiving erroneous data. An extra bit is added to a byte during keyboarding as a means to test accuracy. There are even or odd parity systems and the value of the parity bit will always keep the character's value even or odd depending on the parity system. Using this system of error detection, the values of the bits (1s or 0s) are added sideways including the parity bit value. The sum total will be either odd or even, depending on the parity system being used. If it does not match an error has occurred. This system is not infallible, it two bits exchange values the odd/even parity would remain the same even though it would be erroneous. ***TM 4.04***

The SYSTEM UNIT (pages 65-72) ***TM 4.05***

The system unit is made up of several parts that are often modular and expandable in a computer system. The key parts of a system unit includes:

The **system board** (also called the **motherboard)** usually contains the CPU and main memory chips that are mounted on **carrier packages** that can be plugged into sockets on the system board. The system board also usually has expansion slots.

The **processor chip** is a single silicon chip that contains the computer's CPU. This **microprocessor-- "microscopic processor"** will vary in **word size** (the number of bits in a common unit of information that the computer reads in each cycle of its clock). The most common computer word sizes are 8, 16, 32, and 64. As the word size increases so does speed and power. ***TM 4.06***

The **random access memory (RAM)** is temporary storage and a microcomputer will contain a given amount that the manufacturer deems as standard equipment when the system is purchased. However, knowing how much RAM a system has is very important when working with many of today's popular software applications.
It may be necessary for a user to increase the amount of RAM to accommodate a desired software program. **The read-only memory (ROM)** consists of chips that have built-in instructions (programs) that are made part of the system board by the manufacturer of the computer. These chips cannot be changed by the user and provide the necessary instructions that the computer reads to perform basic operations. **PROM (programmable read-only memory)** allows a user to write instructions on a chip that cannot be changed. **EPROM (erasable programmable read-only** memory) can be erased with special ultraviolet light so that new instructions can be written on it. **EEPROM (electrically erasable programmable read-only memory)** is another type of EPROM.

The **system clock** controls the speed of operation of the computer system. The speed is expressed in **megahertz (MHz** which equals a million cycles of the computers clock in one second). The faster the clock speed, the faster the computer can process information.

The **expansion slots** and **expansion boards** can provide the user with a great amount of versatility. Computers that have expansion slots are known as **open-architecture** computers that permits the user to insert expansion boards. Some common expansion boards include:

expanded memory cards that contain RAM chips that increase the computer's memory.

display adapter cards that allow the user to adapt a color monitor to the computer system, enhance the capabilities of a display with graphics, or improve the resolution quality.

other **"add-on"** cards that can be used to connect peripheral devices such as printers, plotters, phone modems, etc..

additional secondary storage cards that are used to add flexible drives or hard-disk drives to the computer system.

Computers have a limited number of expansion slots, Thus, **multifunction** boards that combine several expansion board features on to one board have been created. Computers that have no expansion slots or capabilities are said to have a **closed-architecture.** *TM 4.07*

The bus line connects the CPU with peripheral devices and other parts of the computer system. It is a data roadway on which bits travel. It resembles a multilane highway. The greater the number of lanes the faster data can travel.
There are three principal bus line architectures:

Industry Standard Architecture (ISA) was first used by IBM as an 8-bit wide data path. This was later widened to 16 bits when the IBM AT was introduced.

Micro Channel Architecture (MCA) was developed when IBM decided to support the new '386 chip with a 32-bit bus line. This architecture does not have the compatibility to allow owners of IBM AT computers to utilize their old expansion boards in this new architecture.
Extended Industry Standard Architecture (EISA) was proposed by nine manufacturers of IBM clones to work as a 32-bit standard that extends from the old ISA standard. The significant difference between EISA and MCA is that the former will allow all existing expansion boards to work with the new 32-bit architecture.

A **Port** is a socket on the outside of the system unit that allows the user to plug other devices into the computer system. A **parallel port** allows lines to be connected that will transmit several bits simultaneously. This type of port is often used by printers and other devices that are physically located close to the computer. A **serial port** transmits bits one after the other on a single communications line. This type of port is used frequently to link equipment that will be physically located at a distance from the computer system. *TM 4.08*

Devices that are outside of the system unit--but not necessarily outside the system cabinet--are called **peripheral devices.**

Usually a peripheral device would include printers, plotters,
phone modems, but it can also include monitors, disk drives, keyboard and any other device that is attached to the computer system.

A LOOK AT THE FUTURE (pages 72-73) *TM 4.09*

There are four new technologies that could make computing faster:

RISC (Reduced Instruction Set Computer) microprocessors will probably begin to replace the present **CISC (Complex Instruction Set Computer)** form of chip design. It is believed by some observers that the RISC chip will triple the performance of the desktop processor every 1 1/2 to 2 years.

New **Superconducting semiconductors** allow electricity to flow through the chips without resistance. This improvement over the present day silcon chip offers the possibility of faster on-off processing to permit lightning-quick computers. This technology was thought to be impractical because the material had to be at extremely low, subzero tempratures. Recent research is now being done on "warm" superconductors.

Optical computing consist of a computer using laser technology which operates with light rather than currents of electricity to represent the on-off codes of data. Light is much faster than electricity so this technology is getting much attention from the computer industry. In January, 1990, Bell Labratories introduced an experimental optical computer.

Neural networks offer a change in the present day **von Neumann architecture** of which most computers used today are based. Instead of having a single processor work on stream of data waiting to be processed, a neural network consists of layers of processors interconnected somewhat like the neurons of a biological nervous system. This arrangement allows data to be transmitted to and from a processor many times faster than the old von Neumann architecture.

These new technologies will allow us to build smaller and faster computers that may permit us to have desktop computers, in the year 2000, that are as powerful as the first supercomputer.

REVIEW QUESTIONS

1. Distinguish between the four kinds of computer systems.

   The following classifications identify the four types of computer systems:

   Microcomputer systems are the smallest form of computers. They range in price from a few hundred up to fifteen thousand dollars. The size of the system allows it to fit nicely on most office desks, some are even portable. The processing capability of these machines can rival minicomputers and easily surpass the power of some older mainframe model computers.

   Minicomputer systems range from several thousand to half million dollars in cost. Generally, they are more powerful, faster at processing, and can store more data than contemporary microcomputers.

   Mainframe computer systems costs several thousand dollars but can easily range into the millions of dollars. They can process millions of instructions per second and can permit a multiple user environment.

   Supermicro computer systems can cost several million dollars. The power of a supercomputer allows it to process over one billion instruction per second.

2. What are the two parts of the CPU and what purpose does each serve?

   The two parts of the CPU are the control unit and the arithmetic-logic unit. The control unit tells the computer system how to carry out instructions and directs the movement of electronic signals between the control unit and the arithmetic-logic unit. The ALU is responsible for arithmetic operations and logical comparisons.

3. What is the purpose of primary storage?

The main memory is the part of the microcomputer that holds data for processing, the program (instructions for processing the data), and information (the processed data) that is waiting to be output or stored in secondary storage.

4. What are registers?

Registers are storage locations in the CPU and ALU. They are a special high-speed staging area that hold data and instructions temporarily during processing. As part of the CPU and ALU the data and instructions can be accessed more quickly than the contents held in primary memory.

5. Describe how the control unit, arithmetic-logic unit, and primary storage work to process information (five steps).

The main memory and the CPU work together in a five step process that turns data into information.

1. Data enters the computer from an input device.
2. The data and the program are held temporarily in memory.
3. The control unit supervises the transfer of data between main memory and the arithmetic-logic unit.
4. The arithmetic-logic unit performs the calculations on the data (processing the data according to the program's instructions).
5. Processed data is sent to an output device.

6a. What is the difference between the decimal system and the binary system?

The decimal system is based on ten digits (0-9). Each place value is a unit of ten for the place value to the immediate right. The first five whole unit place values starting on the right and working left are: 1s, 10s, 100s, 1000s, 10000s, etc.. The binary system works on the same principal, however it is based on a unit of two digits 0 and 1. Each place value is a power of two greater than the place value on the right. The first five whole unit place values starting on the right and working left are: 1s, 2s, 4s, 8s, 16s, etc..

6b. Why is the binary system used in the computer to represent data and instructions?

A computer is made up of electronic circuitry that controls electronic signals that represent data and instructions. These electronic signals have two conditions--ON and OFF. When a program or a user communicates with the computer it must all be translated into ON and Off signals. A convenient number system that can represent the ON and OFF signals is the binary number system. The binary system has only two digits--0 and 1. In the computer, the 0 represents an "off" signal and the 1 represents an "on" signal.

7. What is a bit? A byte?

Each 0 and 1 is called a bit (binary digit) in the binary system. In order to represent numbers, letters, and special characters, bits are combined into groups of eight bits or place values called bytes. A byte is the equivalent of one character, which in many computer systems is one addressable storage location.

8. What is a kilobyte? A megabyte? A gigabyte? A terabyte? What are their abbreviations?

The capacity of memory and secondary storage is expressed in numbers of bytes:

A kilobyte (K, KB, or K-byte) represents 1,024 bytes (two to the tenth power).
A megabyte (MB, or M-byte) represents 1,024,000 bytes.
A gigabyte (GB, or G-byte) represents 1,024,000,000 bytes.
A terabyte (TB, or T-byte) represents 1,024,000,000,000 bytes.

9. What are the names (abbreviations) of the two primary coding schemes for representing letters, numbers, and special characters in binary form?

There are two popular coding schemes that are used in the conversion of a character (that we understand) into a byte of 0s and 1s (that the computer understands). Extended binary coded decimal interchange code (**EBCDIC** pronounced "eb-see-dick") and American standard code for information interchange (**ASCII** pronounced "as-key").

10. What kinds of chips are usually contained on the system board?

The system board contains processor and the memory chips RAM and ROM.The processor is usually contained on a single silicon chip in microcomputers and is called a microprocessor. Some microcomputers have a mathematics coprocessor chip that is subordinate to the main microprocessor and is used to perform fast mathematical computations.

11. What does "word" mean as a measure of the power of a microprocessor?

A computers word size is determined by the size of its processor chip. If a computer has an 8-bit processor, it means that it can process 8 bits with each clock cycle of the computer. A 16-bit processor will process 16 bits with each clock cycle and so on for each of the various processor chips. The more bits that can be processed the greater the computer's capacity to process data quickly and accurately (meaning the computers capacity to work with very large numbers and the number of decimal places for working with small numbers).

12. Distinguish between the RAM and ROM forms of memory?

The RAM (Random Access Memory) is the main memory of the of the computer. It holds the program and data that the CPU processes. The ROM (Read Only Memory) contains instructions for the computer that are built in at the factory and cannot be changed by the user.

13. Why is it important to know the amount of RAM in a microcomputer?

RAM is the temporary working memory of the computer system that holds all the program instructions, raw data, and processed information that the CPU is working on at any given moment. ROM contains the special instructions for detailed computer operations that allows the computer to understand the programs and data that are stored in RAM. Knowing the amount of RAM in a computer is important. Because many popular application programs presently on the market require a minimum amount of RAM to allow the user to successfully operate it.

14. Why are ROM chips also called firmware?

ROM chips are called firmware because the instructions contained on them are "firm" and cannot be changed by the user.

15. What is the purpose of the system clock? How fast is a megahertz?

The system clock controls the speed of operation of the computer system. The speed is expressed in megahertz (MHz which equals a million cycles of the computers clock in one second). The faster the clock speed, the faster the computer can process information.

16. What is an expansion board?

Computers that have expansion slots are known as open-architecture computers that permits the user to insert expansion boards. Computers that have no expansion slots are said to have a closed-architecture. The system board usually contains expansion slots. These slots permit the user to plug in circuit boards that can give the computer increased memory, additional secondary storage, and additional ports to attach a variety of peripheral devices.

17. What kinds of expansion boards are available for what purposes?

Some common expansion boards include:

expanded memory cards that contain RAM chips that increase the computer's memory.

display adapter cards that allow the user to adapt a color monitor to the computer system, enhance the capabilities of a display with graphics, or improve the resolution quality.

additional secondary storage cards that are used to add flexible drives or hard-disk drives to the computer system.

18. What do the bus lines do?

The bus line connects the CPU with other parts and peripheral devices that are part of the computer system. It is like a data roadway where bits of data travel from one part of the computer system to another.

19. Name the three alternative bus architectures.

The three bus line architectures are Industry Standard Architecture (ISA, Micro Channel Architecture (MCA), and Extended Industry Standard Architecture (EISA).

20. What are two types of ports, and what are they used for?

A port is a connecting socket on the outside of the system unit. A cable attached to a peripheral device can plug into this socket. The two types of ports are parallel and serial. Basically, a parallel port is used on devices that are physically close to the system unit.

CHAPTER 5
INPUT and OUTPUT
Chapter at a Glance

I. Input: Keyboard versus Direct Entry

   A. Keyboard Entry
      1. typewriter keys
      2. function keys
      3. numeric keys
      4. directional arrow keys
      5. special purpose keys
         a. Ctrl
         b. Alt
         c. Del
         d. Esc
   B. Terminals
      1. dumb
      2. smart
      3. intelligent
   C. Direct Entry
      1. pointing devices
         a. mouse
         b. touch screen
         c. digitizer
         d. light pen
      2. scanning devices
         a. image scanner
         b. dedicated fax machines
         c. bar code
         d. magnetic-ink character recognition (MICR)
         e. optical-character recognition (OCR)
         f. optical-mark recognition (OMR)
         g. point-of-sale terminal (POS)
         h. touch-tone device
      3. voice input devices
      4. other direct-entry devices

II. Output

   A. Types of Monitors
      1. alphanumeric - monochrome
      2. qualities and features
         a. color
         b. monochrome
         c. resolution

3. graphics
    a. color graphics adapter (CGA)
    b. enhanced graphics adapter (EGA)
    c. video graphics array (VGA)
    d. extended graphics array (XGA)
4. portables
    a. liquid-crystal display (LCD)
    b. electroluminescent (EL)
    c. gas plasma

B. Printers for Microcomputers
1. dot-matrix
2. daisy-wheel
3. laser
4. ink-jet
5. chain

C. Plotters
1. flatbed
2. drum

D. Voice Output

OBJECTIVES

The student should be able to:

1. identify and define the need for keyboard entry devices including dumb, smart, and intelligent terminals.

2. identify and define the purpose of the more common direct-entry devices used with microcomputers.

3. discuss the direct-entry devices used with larger computer systems.

4. identify and contrast the applications of the various types of monitors.

5. identify and contrast the various types of printers.

6. discuss the application of voice-output devices.

VOCABULARY

alphanumeric monitor	drum plotter	laser printer	smart terminal
Alt key	dumb terminal	LCD	softcopy
bar code reader	electroluminescent	light pen	source document
bit-mapping	EGA	MICR	special purpose keys
chain printer	ergonomics	monochrome	speech recognition
CGA	fax machine	monitor	table plotter
Ctrl key	flat panel display	mouse	terminal
daisy wheel	function keys	numeric key pad	touch screen
Del key	gas plasma	OCR	UPC
device	graphics monitor	OMR	VGA
digitizer	hardcopy	pixels	voice-input
directional arrow	image scanner	POS terminal	voice-output
dot matrix printer	ink-jet	printer	wand reader
	intelligent terminal	resolution	XGA

INPUT: KEYBOARD versus DIRECT ENTRY (pages 77-78) *TM 5.01*

Input devices take data that people can read or understand and convert it to a form that can be used by the computer (electronic on and off signals). There are two kinds of input devices: keyboard entry and direct entry.

Keyboard entry devices allow data to be input through a keyboard similar to a typewriter but also has additional keys. These additional keys include typewriter, function, numeric, special-purpose, and directional arrow keys.

Typewriter keys include the letters, numbers, punctuation marks, and special symbols found on most standard typewriters.Usually, a computer user reads from the **source document** and types it to the computer using the keyboard.

Keyboard Entry (pages 78-80)

One of the most common forms of data input is through a keyboard.
Keyboards vary in size, shape and intended purpose. In fact, IBM at one time had over 20 different keyboards to serve all its product lines from top to bottom. Recently, IBM has tried to simplify this situation to a line of 4 keyboards. Most keyboards of today's computer systems will have the following:

Typewriter keys resemble the letters, numbers, punctuation marks, and most special characters found on a typewriter keyboard. This also includes an **enter** or **return** key (found on many electric typewriters) which is used to send a command (commands are stored in the keyboard memory as you type) from the keyboard memory into the computer for processing.

Function keys can be found on the left or right side of the keyboard or often across the top. They are usually labeled as "F1", "F2", and so on, through a sequence of ten or twelve keys. Each of these keys may take on a special purpose or task depending on which application program is currently running. For the programmer, these keys can be programmed to do a variety of tasks that meets the specific needs of the user. Basically, the function keys can save the user keystrokes and the need to do repetitive tasks.

Numeric keys range from 0 through 9 and are found across the top row of the typewriter keyboard. Most computer keyboards also have a numeric key pad on the right hand side of the keyboard (it resembles the key pad found on most adding machines). The keys on the key pad can perform dual purposes for tasks involving numbers and as a method of moving the screen's cursor.

The **directional arrow keys** are used for moving the cursor. Many present day keyboards have two sets of directional arrow keys: one on the numeric key pad and a separate set that are solely direction keys. Having two sets allows one to use the numeric key pad solely for numeric tasks which circumvents the need to toggle between the numeric key pad and directional arrow keys.

Special-purpose keys would include **Ctrl (control), Del (delete), Alt (alternate)**, etc.. Some of these keys can perform editing tasks (like Del) while others are only used in conjunction with other keys (like Ctrl and Alt) to perform tasks or special operations.

Terminals (pages 79-80) ***TM 5.02***

A **terminal** is an input/output device that includes a keyboard, a monitor, and a communications link into a CPU. The three types of terminals are **dumb, smart**, and **intelligent.**

A **dumb terminal** does not have its own processor, thus, cannot process data independently. It is used solely as a means of inputting data and receiving information.

A **smart terminal** has limited memory capacity to allow users to perform some editing or verification of data before it is sent to the CPU for processing.

An **intelligent terminal** is a computer in its own right, meaning that it has a CPU, primary and secondary storage, and software for processing data. Usually a microcomputer serves as the intelligent terminal.

Direct Entry (pages 80-84) ***TM 5.03***

Direct entry utilizes devices that create machine-readable data on a medium that can be input directly into a CPU. This is a very efficient means of input that is economical and less prone to error than if the data had to be key entered. Direct-entry devices include the following:

A **mouse** which is a device for moving the cursor or pointer around the screen. The mouse has a track ball that rolls on a flat surface. The cursor will be oriented to the point where the mouse is placed on the flat surface, as the mouse moves up, down, left, right, etc. the cursor will move in the same respective direction. The mouse has selection buttons that allow the user to move the cursor to menu items on the screen then select them using a button.

A **touch screen** is a monitor screen covered with a plastic layer. Behind this layer of plastic are crisscrossed beams of infrared light. When the screen is touched the position of the contact is determined by the break in the coordinates of the infrared beams. Touch screens are easy to use and have many applications.

A **digitizer** is a device used to convert an image into digital data that is input into the computer. The digitizer is moved or scanned over a drawing or photograph that will send the digital data to the screen, a printer, or to a disk for storage.

A **light pen** is used with a special type of monitor that allows for the entering or modification of data that appears on the screen. When the light pen is placed against or near the screen's surface, it activates a photoelectric circuit.

Scanning Devices (pages 81-83) ***TM 5.04***

An **image scanner** (also called a **bit-mapping device**) converts images into electronic signals that can be processed and stored in the computer. The process identifies pictures or different typefaces by scanning the image with light and breaking it into light and dark dots.

Facsimile transmission machines (fax machines) have become an extremely popular office machine. It provides all the convenience of a copy machine but takes a step further by sending the copy electronically through the telephone lines to a receiving fax machine. *TM 5.06*

Magnetic-ink character recognition (MICR) is used by banks to read the numbers found at the bottom of check. A reader/sorter is the direct-entry device used to read the characters that are made of ink containing magnetized particles.

Optical-character recognition (OCR) uses special characters that can be read with a wand reader or some other type of OCR device. The characters have a specific formation that can not have any deviation if the OCR reader is to correctly read the character. This method of data entry is often used by retail stores and utility companies.

Optical-mark recognition (OMR) senses the absence or presence of a mark (usually made with a pencil). This method of data entry is often used by colleges for test taking and is also known as mark sensing.

Voice Input Device (pages 83-84)

A **voice-input device** (also known as **speech-recognition** or **voice-recognition devices**) converts human speech into digital code.
Because of the countless qualities that makes each individual's voice different, the system must be calibrated to the user's voice. This is accomplished by having the user speak into the device to store the unique speech patterns. These patterns are later used to match the spoken data that is entered by this same user. Problems can occur if the user develops a cold. Some systems can recognize the spoken word of many people, however, the vocabulary is limited with these systems. The advantage of a voice-input system is that it frees the users hands for other tasks.

Other Direct-Entry Devices

A **point-of-sale terminal (POS)** has a keyboard, a screen, and a printer.They are used like a cash register in various types of stores. They often have a **bar code reader** or **scanner** that can read the zebra-striped Universal Product Code (UPC). This code can be matched with the product in the computer and create a receipt with the product name and price. It will also make adjustments to the inventory as the product is sold.

A **touch-tone device** uses the telephone lines to send data to a computer. A **card dialer** can be used to determine an individual's credit card status by running the credit card through the dialer. Signals are sent via the telephone line to the computer and returns a status report that the user can read from the card dialer.

OUTPUT (page 84)

An **output device** is designed to give the user feedback in an understandable form. The following devices are some commonly used output devices:

Types of Monitors (pages 84-86) *TM 5.05*

Monitors - classified as alphanumeric, graphic, and flat-panel displays. **Resolution** is determined by the density of **pixels** (individual dots or picture elements that form the screen image) on the screen.

An **alphanumeric monitor** displays letters, numbers, punctuation and a variety of special characters found on computer keyboards. They are very often a **monochrome** monitor (a display of only one color, usually amber, green, or white against a dark or solid background).

A **flat-panel display** is classified into **liquid-crystal (LCD)** or **electroluminescent (EL)**. The flat-panel display differs from the graphic and alphanumeric monitors in that the latter two are a vacuum tube technology. The LCD uses liquid crystal molecules that have their optical properties altered by an electric field. An EL is able to emit light when it is electrically charged.

A **graphics monitor** displays alphanumeric characters and picture graphic images. The graphics monitor has gone through three stages:

CGA (color graphics adapter), was introduced by IBM and is a circuit board that can be inserted into the computer. This board gave the computer color graphic capability provided a monochrome monitor was connected to it. In a monochrome mode its resolution quality was 640 x 320 pixels (very good for word processing). In a color mode, the CGA resolution for four colors was 320 x 200 pixels (the text display would be of a course grain and not as clear to read).

EGA (enhanced graphics adapter), has a resolution of 640 x 350 pixels supporting 16 colors. If the user purchased this board it usually meant that they also had to upgrade the monitor. This board is still very popular today.

VGA (video graphics adapter), in a text mode, this board has a resolution of 720 x 400 pixels. With 16 colors this board will produce resolution of 640 x 480 pixels. It is starting to gain in popularity and has many applications because of its clarity.

XGA (extended graphics array) has a resolution up to 1024 x 768 pixels. It displays up to 256 colors under normal conditions. With special enhancements, it can display up to 65,536 colors. This monitor is becoming the standard for PS/2 486 processor technology.

Printers for Microcomputers (pages 86-89) ***TM 5.06***

Printers are used to create a **hardcopy** (output on paper). Some features to consider when comparing printers include bidirectional printing, friction feed, tractor feed, type styles, color, and the ability to print graphics.

The popular kinds of printers used with microcomputers include dot-matrix, daisy-wheel, laser, ink-jet, and chain.

The **dot-matrix** printer forms its image using a matrix of pins that travel horizontally back and forth as the paper is fed through vertically. Each pin has the capability of forming a dot on the paper. These dots are configured together to form images on the paper much like a basketball scoreboard has individual lights to form numbers. The features of dot-matrix printers include reliability, relatively fast printing, capability to print near letter-quality documents, ability to make a graphic image, and printing color using a multi-color ribbon. The dot-matrix printer is the most popular printer used with microcomputers.

The **daisy-wheel** printer uses the same principal as a typewriter. It has a type set for each character on the keyboard and usually a few special characters that a particular individual might need. A hammer strikes the type set that hits the ribbon against the paper to form the image. It is called a daisy-wheel because the type set is placed on a wheel with individual spokes resembling a daisy. Each spoke has a raised character at the end of it to form the typed image. The daisy-wheel printer features very high quality print and the ability to exchange wheels for

different fonts of print. Speed, inability to print quality graphics, and initial cost that are usually higher than the cost of a dot-matrix printer are all disadvantages of the daisy-wheel printer.

The **laser-printer** creates an image so good that it has spawned the new industry of desk-top publishing. Used with special software, the laser printer can merge text and graphics that has a quality that compares with professional typesetters and graphic artist. The laser-printer bounces a laser beam on to a drum forming a dotlike image (like a dot-matrix printer). This image is a magnetically charged inklike toner that is then transferred from the drum to a special paper. A final process of heat makes the characters adhere to the paper.

The **ink-jet** printer sprays droplets of ink that produces a high quality image in a variety of colors. This type of printer allows the user to duplicate the color image that appears on the screen.

The **chain printer** is a very expensive, high-speed printer designed to work with mainframe and minicomputers. With the development of the microcomputer network, this technology can be cost justified for the microcomputer environment. It has several sets of characters hooked together on a printing chain, which revolves in front of the ribbon and paper. Hammers are aligned with each position. When a character passes by, the hammer in a particular position will strike to produce the desired character on the paper. Chain printers reach speeds of 3000 lines per minute and are very reliable.

Some general qualities to note about microcomputer printers include:

Bidirectional - meaning the print head is printing as it moves in both directions horizontally across the printer.

Tractor feed - helps reduce misalignment of continuous form paper. It is a mechanism of sprockets that the holes from continuous form paper mesh into so the printer can advance the paper quite rapidly.

Type styles - are various fonts that can be changed through the use of software and the right kind of printer hardware. Some printers have printing elements that can be exchanged to allow for a different type style. Not all printers have this capability.

Shared use - is the ability to have more than one computer share expensive resources. Dot-matrix and daisy-wheel printers are often used with a single microcomputer. Ink-jet, laser, and chain printers are often linked to several microcomputers through a communications network. This is a more practical arrangement considering the expense of these type of printers.

Plotters (pages 89-90) *TM 5.07*

A **plotter** is another type of hardcopy producing device. Its purpose is intended more for producing bar charts, maps, and architectural drawings, and even three dimensional illustrations. Some plotters can also handle large size documents. They come in two types: flatbed and drum.

The **flatbed (table plotter)** has stationary paper with moving pens that draw the image.

The **drum plotter** has pens that move horizontally and paper that roll vertically on a drum. When the paper and pen move at the same time, a curved line is drawn.

The **electrostatic plotter** uses charges making little dots on specially treated paper to produce an image. Electrostatic plotters produce high-resolution images in less time than conventional pen plotters.

Voice Output (page 90)

A **voice-output** device has prerecorded vocalized sounds that the computer can activate to resemble human speech. This synthesized speech activated by a computer has many applications and is growing in use.

A Look at the Future (pages 90-91)

The input/output process will undergo some significant transition as:
* portable computers become smaller.
* keyboards are replaced with devices that convert handwritten text into electronic data.
* sublaptops can have wireless communications with stationary computers.
* voice-recognition technology becomes more sophisticated.
* one-inch display screens are developed for hand-held computers.
* flat-panel, full-color screens replace CRTs.
* graphics become more lifelike.
* microcomputers and television technologies merge allowing us to watch TV on our computer--but, also to freeze video images to create still images that can be stored.

REVIEW QUESTIONS

1. What are the differences between keyboard entry and direct entry as forms of input?

   Data that is entered from a typewriter like keyboard is referred to as keyboard entry data. Data that is produced in a computer-processable form as it is entered into the computer is referred to as direct entry data.

2. What is a POS terminal? What are the two input devices on it that represent the two methods of inputting data?

   A point-of-sale terminal has a keyboard, a screen, and a printer. They are used like a cash register in various types of stores. They often have a bar code reader or scanner that can read the zebra-striped Universal Product Code. This code can be matched with the product in the computer and create a receipt with the product name and price.

3. What are the four kinds of keys on a keyboard?

   The four kinds of keys on a keyboard are the typewriter keys (letters, numbers, special characters, and the return), function keys (labeled F1, F2 and so on), numeric keys (positioned together to form a numeric keypad), and special purpose keys (directional arrow, Esc, Ctrl, Del, and Ins).

4. Distinguish among the three kinds of terminals: dumb, smart, and intelligent.

   A dumb terminal does not have its own processor, thus, cannot process data independently. It is used solely as a means of inputting data and receiving information.

   A smart terminal has limited memory capacity to allow users to perform some editing or verification of data before it is sent to the CPU for processing.

   An intelligent terminal is a computer in its own right, meaning that it has a CPU, primary and secondary storage, and software for processing data. Usually a microcomputer serves as the intelligent terminal.

5. List some direct-entry input devices.

   Some typical direct entry devices include the mouse, digitizer, light pen, image scanner, and voice-input devices. In addition there scanners that can read various forms of marks and images that include the magnetic-ink character recognition, optical-character recognition, optical-mark recognition, and point-of-sale terminals that use bar code readers. Touch-tone dialers can also be considered a direct-entry device that allows an individual to input data into a computer using a touch tone telephone.

6. How does a mouse work?

   A mouse moves the cursor or pointer around the screen by a means of using a track ball that rolls on a flat surface. The cursor will be oriented to the point where the mouse is placed on the flat surface, as the mouse moves up, down, left, right, etc. the cursor will move in the same respective direction. The mouse has selection buttons that allow the user to move the cursor to menu items on the screen then select them using a button.

7. What input device recognizes images and converts them into electronic signals?

   Digitizer

8. How does an image scanner work?

   An image scanner scans an image with light and breaks it into light and dark dots that can then be converted to digital code for storage.

9. Which direct-entry input device is particularly helpful to certain disabled people?

   The voice-input devices enable users to keep their hands free for other tasks. Thus, they are an advantage for disabled people to use as a means od data input.

10. List four output devices.

    Output devices would include monitors, printers, plotters, and voice.

11. What uses are monochrome monitors best suited for?

Monochrome monitors are best suited for word processing and other text applications because of the higher resolution quality in this mode and possibly less eyestrain.

12. What are the three kinds of flat-panel displays used with portable computers?

The flat panel displays used with portable computers include: liquid-crystal display (LCD), electroluminescent (EL), and gas-plasma display.

13. What are pixels? What do they have to do with screen resolution?

Resolution is determined by the density of pixels on the screen. A pixel is an individual dot or picture element when combined with other pixels form a screen image. A matrix of 200 row by 300 column dots is a medium resolution quality, 200 by 640 is considered a high resolution display.

14. Explain how a dot-matrix printer works.

The dot-matrix printer forms its image using a matrix of pins that travel horizontally back and forth as the paper is fed through vertically. Each pin has the capability of forming a dot on the paper. These dots are configured together to form images on the paper much like a basketball scoreboard has individual lights to form numbers.

15. Explain how a daisy-wheel printer works.

The daisy-wheel printer uses the same principal as a typewriter. It has a type set for each character on the keyboard and usually a few special characters that a particular individual might need. A hammer strikes the type set that hits the ribbon against the paper to form the image. It is called a daisy-wheel because the type set is placed on a wheel with individual spokes resembling a daisy. Each spoke has a character at the end to form the typed image.

16. Describe how an ink-jet printer operates.

The ink-jet printer sprays droplets of ink on the paper that produces a high quality image and offers a variety of colors.

17. State how a laser printer works.

The laser-printer bounces a laser beam on to a drum forming a dotlike image (like a dot-matrix printer). This image is a magnetically charged inklike toner that is then transferred from the drum to a special paper. A final process of heat makes the characters adhere to the paper.

18. What can a plotter do that a printer cannot?

A plotter is another type of hardcopy producing device. Its purpose is intended more for producing bar charts, maps, and architectural drawings, and even three dimensional illustrations. Some plotters can also handle large size documents that could cover a large wall. It makes its image with pens that contact the paper.

19. What is the difference between a flatbed plotter and a drum plotter?

The flatbed (table plotter) has stationary paper with moving pens that draw the image. The drum plotter has pens that move horizontally and paper that roll vertically on a drum. When the paper and pen move at the same time, a curved line is drawn.

20. Is voice output more difficult to engineer than voice input?

Synthesized speech activated by a computer has many applications and is much easier to use that voice-input because humans have a discriminating ability to understand the spoken word from a variety of sources. A computer has difficulty duplicating this human ability because of the countless variables that are involved when humans speak. These variables include dialect, intonation, inflection, pitch, etc..

CHAPTER 6
SECONDARY STORAGE
Chapter at a Glance

I. Direct Access Versus Sequential Access

II. How Data is Organized

- A. Character
- B. Field
- C. Record
- D. File
- E. Database

III. Four Kinds of Secondary Storage

- A. Diskettes
  1. device used for storage
  2. direct access
  3. construction
  4. parts of a diskette
  5. care and treatment
- B. Hard Disk
  1. internal
  2. hard-disk cartridge
  3. hard-disk pack
  4. access time
     - a. seek time
     - b. head switching time
     - c. rotational delay time
     - d. data transfer time
  5. head crash
- C. Optical Disks
  1. CD-ROM
  2. WORM
  3. reusable
- D. Magnetic Tape
  1. sequential access
  2. tape streamers
  3. tape reels
     - a. IRG
     - b. IBG
     - c. tape libraries

IV. A Look at the Future of Secondary Storage

OBJECTIVES

The student should be able to:

1. differentiate direct access with sequential access storage.

2. define how data is organized by describing its classifications.

3. describe how diskettes and disk drives allow for the storage and retrieval of data.

4. discuss how hard disks and disk packs operate.

5. discuss the application of mass storage and optical disks.

6. describe the purpose and the operation of magnetic tape storage.

VOCABULARY

access arm
access time
bit
byte
CD-ROM
character
data transfer time
database
direct access
disk drive
diskette
drive gate
field
file
floppies
formatting
hard disk
hard disk cartridge
head crash
head switching time
IBG
IRG
magnetic tape
mass storage
mylar
nonvolatile memory
optical disk
random access
read/write head
record
rotational delay time
search
secondary storage
sectors
seek time
sequential access
soft-sector diskette
tape library
tape streamer
tracks
volatile
WORM
write protect notch
write-enable ring

DIRECT ACCESS VERSUS SEQUENTIAL ACCESS (page 95)

A disk unit is also known as a **random access**, **direct access**, and as an **online system.** It is much faster than a sequential access system. It allows the computer to move to the specific item of data for retrieval at any moment. A **sequential access system** uses a tape storage medium. To access data, the computer must reel through the tape to find the desired data. This method takes much longer but has the added benefit of providing unlimited storage capacity. In addition, tape access is more secure because it is not directly accessible to the computer and cannot be modified by someone who gains access to the computer's CPU.

DATA ORGANIZATION (page 96) *TM 6.01 & TM 6.02*

The organization of data can be described as a hierarchy that we will discuss from the smallest to the largest unit of classification.

A **character (byte)** is a single letter, number, or special character.

A **field** is a set of related characters. It is an item of data such as a telephone number, street address, etc..

A **record** is a collection of related fields. A group of data items that have a common relationship as a group. An inventory file might have a part number, cost, number on hand, and the distributor's name and address as the related fields making up each record in the file.

A **file** is a collection of related records. In our example, all the inventory items would collectively make the data file.

A **database** is a collection of related files. If we have a file that keeps track of customers and the parts in our inventory that they purchases, it is a likely event for us to cross reference this file with our inventory file. Database software allows the user merge, interact, and cross reference multiple files that have some common link.

FOUR KINDS OF SECONDARY STORAGE (page 97) ***TM 6.03***

Microcomputers have four forms of secondary storage: **diskette, hard disk, optical disk,** and **magnetic tape.**

DISKETTES (pages 97-101) ***TM 6.04 & TM 6.05***

Diskettes (disks) come in the 5 1/4-inch size or the 3 1/2-inch size. They can also be called a **flexible disk, floppy disk**, or simply **disk.** The disk that holds the data is enclosed in an outer jacket that is somewhat flexible in the 5 1/4-inch size, but is very rigid in the 3 1/2-inch size. The disk is made of a plastic called **mylar** that is coated with a metallic substance (usually iron oxide). The disk drive can arrange the disk's metallic surface into electromagnetic impulses. These impulses represent an "on" (magnetized spot) or an "off" (non-magnetized spot) signal on the disk. These spots are arranged in sets, typically using the ASCII data representation code.The **disk drive** has a slot that allows the user to slide in a diskette and close the **drive gate** (which places the read/write head in position allowing it to touch the disk surface). A spindle fits into the disk's center hole and is ready to spin the inner disk surface (at about 300 rpm) when the disk drive is activated by the computer. Nearly all microcomputers have at least one disk drive (often built into the system unit) as a means to initially enter software programs into the computer.

The **read/write head** is on an **access arm** that moves back and forth to position over a specific track on the disk surface (this is called a **seek** operation). At the same time, the spindle is rotating the disk to the desired location to read or write the data (this is called the **search** operation).

For convenience and additional storage capacity, many systems have two floppy disk drive units. The units are identified as **drive A** (for the first drive unit, usually found on the left or top if there is more than one drive), and **drive B** (for the second drive unit, usually found on the right or bottom of drive A).

The **parts of a disk** include :
* **tracks** - concentric circles where data may be placed
* **sectors** - divide the tracks into pie-shaped sections
(The fields of data are organized according to the tracks and sectors on the disk, giving the data a specific addressable location on the disk.)
* **write-protect notch** - allows the reading of data but prevents the writing of data on the disk surface when the notch is covered. (This avoids the possibility of writing over or erasing data already contained on the disk.)

The types of disks vary depending upon the type of disk drive. Some disk drives can read only one side of the disk, while others can use both sides of the disk. In addition, the disk drive's ability to compact the data relates to the density; which can be single, double, or even quad density. The disk label will usually indicates the type of disk, such as "DS,DD" or "2S/2D" (for double sided, double density). The capacity of a disks will vary depending on its density, single or double side, and its size. However, size can be deceiving. A typical DS,DD 5 1/4-inch floppy can hold about 360KB, but a DS,DD 3 1/2-inch version can hold about 720KB (high density 1.4 megabytes).

HARD DISK (pages 101-104)

A **hard disk** is a factory sealed unit that has one or more metallic platters, an access arm, and read/write heads. The operation and size is much the same as a floppy disk drive unit (they can come in a 5 1/4-inch or a 3 1/2-inch diameter disk). Because the integrity of the disk surface is greater (due to the lack of access and outside contaminants), a greater storage capacity is possible. In addition the access time is faster and more reliable than a floppy disk system. A major difference with a hard disk system is that the read/write head never touches the disk surface. The tolerance between the head and the disk is so thin that a particle of smoke would not fit between them. The hard disk unit is a sensitive device, if the head were to ever touch the disk surface it would ruin the unit. It is very important to know the care of your system, (especially when you plan to move it) this information can be obtained from the owners manual.

A **hard-disk cartridge** can be removed, allowing for unlimited storage capacity (limited only by the number of cartridges one makes available to the system) along with fast access.

A **disk pack** is predominately used with mini and mainframe computers. It contains several platters stacked above one another. These platters are spaced to allow the access arm and the read/write heads (one set for the top side and one set for the bottom side) to move in and out. *TM 6.06*

OPTICAL DISK (pages 105-106) *TM 6.07*

An **optical disk** can hold up to 700 megabytes of data. This technology makes an immense amount of information available to a microcomputer. An optical disk employs a laser beam which burns tiny pits representing data into the surface of a plastic or metal disk. The disk sizes vary in diameters of 3 1/2, 4 3/4, 5 1/4, 8, 12, and 14 inches. The kinds of optical disks available are:

Read-only: A plastic disk with data represented as small pits in the disk surface. Data is imprinted on them by the manufacturer holding 540 to 738 MB of data. The user can only read this data off of the disk, data cannot be stored on the optical disk by the user. The most popular type of read-only optical disk is **CD-ROM (compact disk-read only memory)**.

Write-once: A metal-film disk surface recorded on by lasers. The user can write data on a surface area only once; it cannot be erased from that area once it is written. The data written on this disk surface (it is often done by the manufacturer), does not deteriorate and can be read many times. This type of optical disk is called **WORM (write once, read many)** storing between 122 and 6400 MB of data.

during pathname analysis, the corresponding request is sent to associated port (see § 4.4.1.). Figure 14 illustrates the interconnection of two machines, `pipo` and `piano`, using symbolic ports. In the example, the pathname `/fs/piano/usr/fred` refers to the same directory from both machines because `/fs/piano` is a symbolic port to the File Manager of the `piano` root file system.

CHORUS/MIX also implements **symbolic links** à la BSD. When combined with symbolic ports, this functionality provides an extremely powerful and flexible **network file system**.

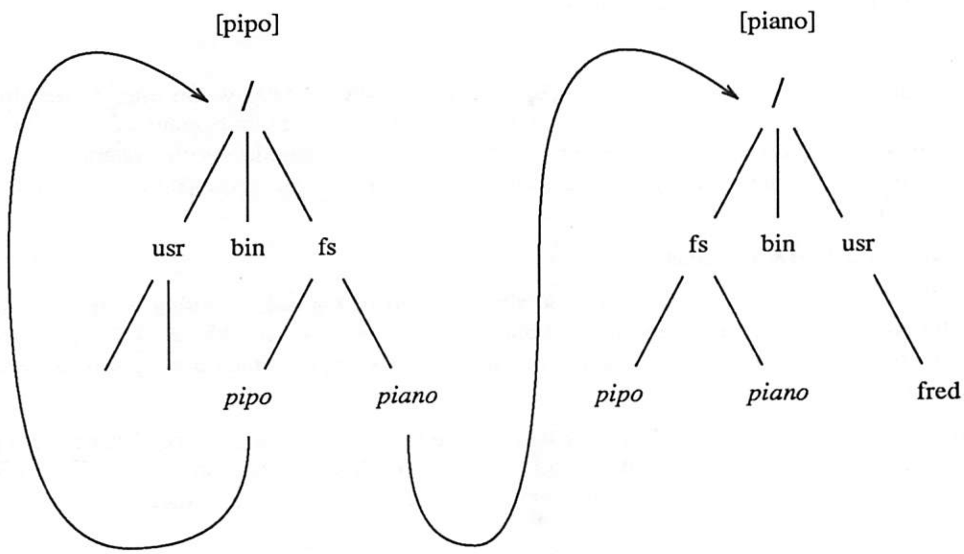


Figure 14. – File Trees Interconnection

4.2.4 Socket Manager

The **Socket Manager** implements UNIX 4.3 BSD socket management for the Internet address family. Socket Managers are only loaded onto sites that have network access.

4.2.5 Device Manager

Devices such as tty's and pseudo-tty's, bitmaps, tapes, and network interfaces are managed by **Device Managers.** There may be several Device Managers per site, depending upon site device requirements, which may be dynamically loaded or unloaded. Software configurations can be adjusted to suit the local hardware configuration or the needs of the user community.

A CHORUS IPC-based facility is used to replace the `cdevsw` table found in traditional UNIX systems. During initialization, the Device Manager sends its port and the major numbers of devices that it manages to the File Manager. When these major numbers are encountered during pathname analysis for an `open` system call, the request is sent to the associated port.

4.2.6 IPC Manager

The CHORUS/MIX IPC Manager (IPCM) provides user services equivalent to those supplied by a UNIX System V.3.2 kernel on inter process communication (IPC). These services include: messages, semaphores and shared memory. The IPCM was built from UNIX System V.3.2 IPC code.

The IPCM interacts with the PM, the KM (Key Manager, see below) and the FM. The PM is the main IPCM's "client": it transfers user system call requests and arguments to the IPCM, which handles them and returns values according to UNIX IPC semantics. Note that since they do not have any device connection, IPCM's may be present on any CHORUS/MIX site.

For shared memory operation, when a request is made to create a new shared segment, the IPCM requests service from the Object Manager to initialize a new segment capability. This capability will be used by processes trying to attach the shared segment to their address space, using the *rgnMap*() system call.

4.2.7 Key Manager

The CHORUS/MiX Key Manager (KM) is an internal CHORUS/MiX server. This means it does not provide any service to user programs, but accepts requests from the PM and the IPCM, in the context of the UNIX System V.3.2 IPC services.

The Key Manager creates and maintains a mapping between user provided *keys* and CHORUS/MiX internal *descriptors*. It ensures the uniqueness and coherence of the mapping accross a distributed system. There must be only one KM on a CHORUS/MiX system.

4.2.8 User Defined Servers

The homogeneity of server interfaces provided by the CHORUS IPC allows system users to develop new servers and to integrate them into the system as user actors. One of the main benefits of this architecture is that it provides a powerful and convenient platform for experimentation with system servers. For example, new file management strategies or fault-tolerant servers can be developed and tested as a user level utility without disturbing a running system.

4.3 Structure of a UNIX Process

A traditional UNIX process can be viewed as a single thread of control executing within one address space. Each UNIX process is, therefore, mapped onto a single CHORUS actor whose UNIX system context is managed by the Process Manager. The actor's address space comprises memory regions for text, data, and execution stacks.

In addition, the Process Manager attaches a **control port** to each UNIX process actor. This control port is not visible to the user of that process. A control thread in the Process Manager is dedicated to receive and proceed all the requests on this port. This control thread executing within process contexts has two main properties:

- It shares the process address space and can easily access and modify the memory of the process in order to perform text, data, and stack manipulations during signal delivery or during debugging.
- It is ready to handle asynchronous events, such as signals, which are received by the process. These events are implemented as CHORUS messages received on the control port (Figure 15).

Allowing for multi-threaded processes has impacted the implementation of each process system context. The UNIX system context attached to a process has been split into two system contexts: a process context (`Proc`) (Table 1) and a u_thread context (`UThread`) (Table 2).

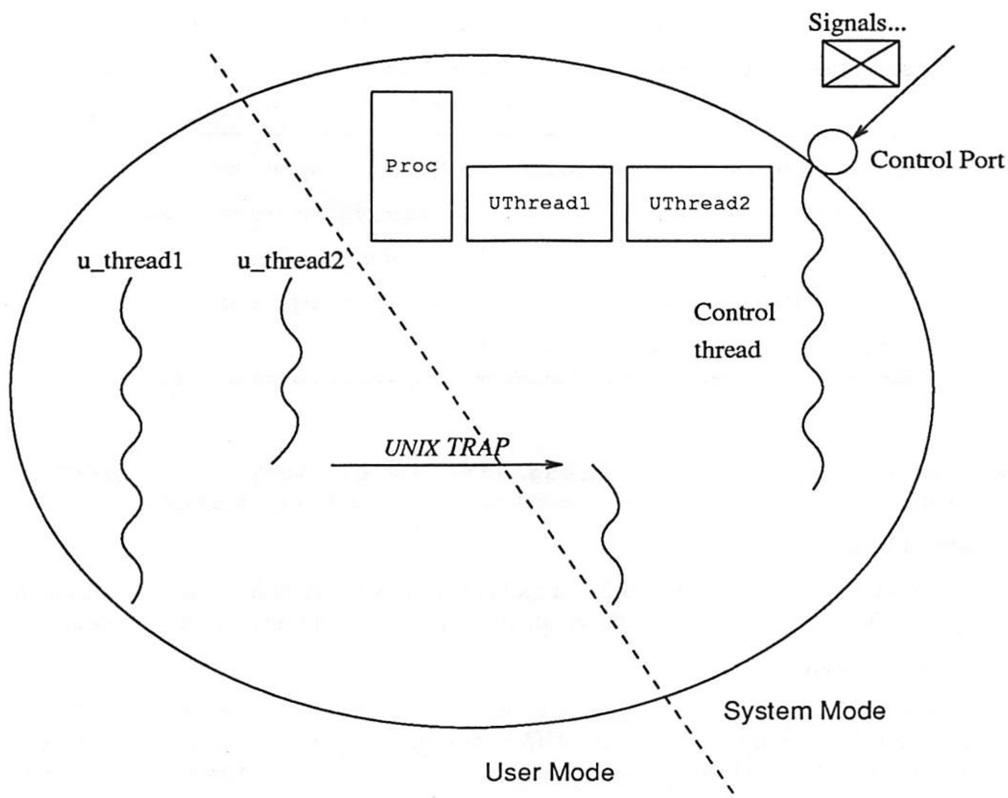


Figure 15. – UNIX Process as a CHORUS Actor

Table 1. – Process Context

Proc Context	
Actor implementing the Process	*actor name, actor priority, ...*
Unique Identifiers (UI)	*PID, PGRP, PPID, ...*
Protection Identifiers	*real user id, effective user id, ...*
Ports	*control port, parent control port, ...*
Memory Context	*text, data, stack, ...*
Child Context	*SIGCLD handler, creation site, ...*
File Context	*root and current directory, open files, ...*
Time Context	*user time, child time, ...*
Control Context	*debugger port, control thread descriptor, ...*
UThreads	*list of process' UThread contexts,*
Semaphore	*for concurrent access to* `Proc` *Context.*
IPC context	*UNIX messages, semaphores, shared memory.*

Table 2. – UThread Context

U-thread Context	
Thread implementing the u_thread	*thread descriptor, priority, ...*
Owner Process	*owner process proc descriptor*
Signal Context	*signal handlers, ...*
System Call Context	*system call arguments, ...*
Machine execution Context	

The two system contexts, `Proc` and `U_thread`, are maintained by the Process Manager of the current process execution site. These contexts are accessed uniquely by the Process Manager.

4.3.1 Process Identifiers

Each process is uniquely designated by a 32 bit global PID which results from the concatenation of two 16 bit integers. These integers consist of the creation-site id and a traditional UNIX process id.

4.3.2 Process Execution Site

As an extension to the standard UNIX process semantics, CHORUS/MiX maintains a notion of the child process creation site for each process. This site identifies the target site to which the `exec` operation is applied. By default, the child process creation site is set to the site on which the process currently resides.

4.3.3 Process Control

The Process Manager attaches a **control port**[4] to each CHORUS/MiX process. A dedicated thread in the Process Manager (called the control thread) listens on the all PM's ports for process management directives from CHORUS/MiX subsystem servers. These **control messages** include: UNIX signal messages, debugging messages, and process exit messages. Only the control thread may receive messages on the control port; the control port is not exported to the CHORUS/MiX process.

When the process is the target of a `kill` system call or of a keyboard-generated signal, a signal delivery message is sent to its process control port.

When a CHORUS/MiX process performs a `ptrace` system call to initiate a debugging session, the PM creates a debug port and sends it to the debugger. All `ptrace` functions performed by the debugger are translated into messages and sent to the debug port. Since these interactions are based on CHORUS IPC, the process and its debugger may reside on different sites.

A process context contains the control port of its parent. When the process exits, an exit status message is sent to its parent's control port. This status information is stored in the parent's process context where it can be retrieved using the `wait` system call.

4.3.4 Process Resources

A process refers to all server-managed resources by using their server-provided capabilities. Open files and devices, current and root directories, and text and data segments are notable examples of such resources. These capabilities are standard CHORUS capabilities and therefore can be used in conjuction with the CHORUS mapper protocol, if appropriate.

4. A single control port is required for CHORUS/MiX processes; multiple control ports may be needed by other subsystem implementations.

For example, opening a device file associates the capability provided by the device's server with a UNIX file descriptor. The capability will be constructed from the port of the Device Manager and a reference to the device within the manager.

All subsequent system calls pertaining to that device will be translated into messages and sent directly to the appropriate server. There is no need to locate the server again.

4.4 Two Examples

4.4.1 File Access

Current and root directories are represented by capabilities in the UNIX context of a process. When an `open` request is issued, the open routine of the Process Manager looks for a free file descriptor, builds a message containing the pathname of the file to be opened, and sends this message to the port of the server managing the current or the root directory, depending on whether the pathname is absolute or relative (Figure 16 [1]).

Suppose that the pathname of the file is `/fs/piano/usr/fred/myfile` and `/fs/piano` is the symbolic port of a File Manager running on a site named `piano`. This pathname will be sent to the File Manager containing the root directory of the process (the `pipo` File Manager in the example). That File Manager will start the analysis of the pathname, discover that `piano` is a symbolic port, and propogate the message with the unanalyzed portion of the pathname (`/usr/fred/myfile`), to the symbolic port.

The `piano` File Manager will receive the message, complete the analysis of the pathname, open the file, build the associated capability and send the capability back to the client process that issued the `open` request (Figure 16 [3]).

Any subsequent request on that open file will be sent directly to the File Manager on `piano`. The `pipo` File Manager will not be involved in further interacations.

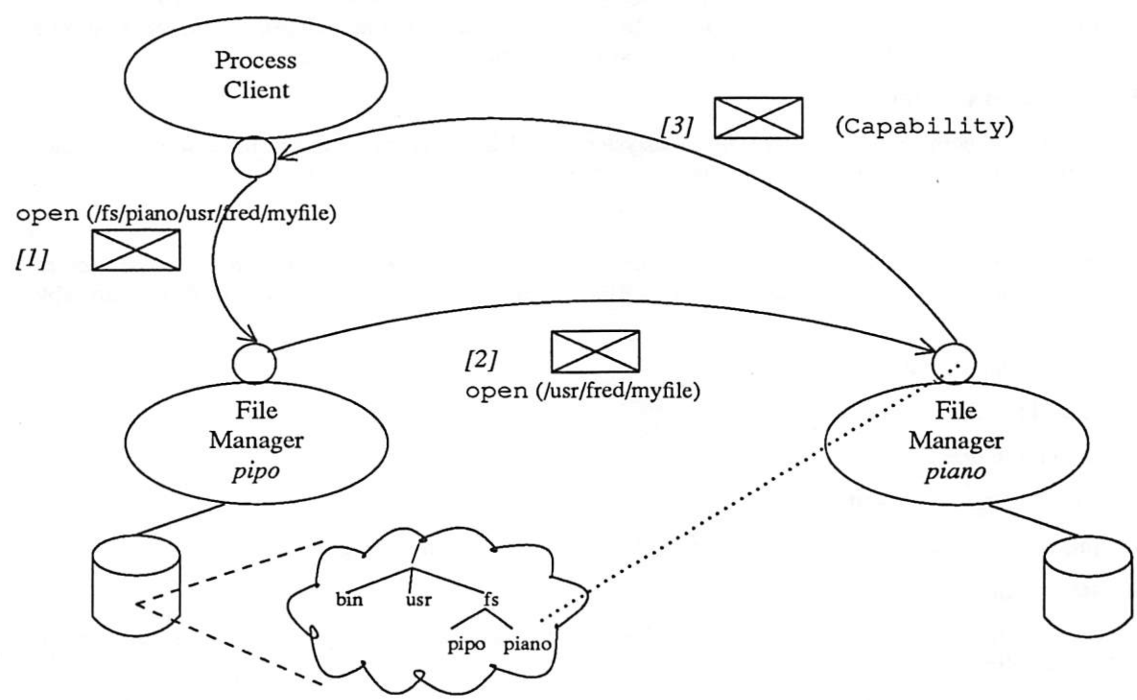


Figure 16. – File Access

4.4.2 Remote Exec

This description of the remote `exec` algorithm will illustrate all the interactions between the UNIX subsystem servers and the process control threads. To simplify the description, error cases are not handled in this algorithm.

1. the calling u_thread performs a trap handled by the local Process Manager. The PM will :
   a. determine whether the pathname is relative or absolute, and therefore whether to use the File Manager of the root directory or the current directory;
   b. invoke by RPC the File Manager to translate the binary file pathname into two capabilities, used later to map text and data into the process address space;
   c. if the child execution site is different from the current execution site, test the child execution site for validity;
   d. prepare a request with:
      - the `Proc` and `UThread` contexts of the calling u_thread;
      - the arguments and environment given as `exec` parameters;
      - all information returned by the File Manager that characterizes the binary file.
   e. perform an RPC to the Process Manager of the target creation site by means of the Process Manager port group.
2. The Process Manager of the remote creation site receives the request. One of its threads initializes a `Proc` context for the new process using information such as PID's, elapsed time, and open file capabilities, contained within this message. The Process Manager creates new CHORUS entities which implement the process, such as an actor, a control thread, a control port, and memory regions. It then proceeds with UNIX process initialization:
   - it installs arguments and the environment strings in the process address space;
   - it sends close messages to appropriate File Managers, Socket Managers and Device Managers to close open files marked ''close-on-exec'';
   - it creates one u_thread, which will start executing the new program (after `exec`, all processes are initially single-threaded). It initializes the signal context of the created u_thread using the signal context of the calling u_thread which is present in the **request message.**
   - it sends a reply message to the u_thread that originally invoked the `exec` system call.
3. The calling u_thread receives the reply message, frees the `Proc` and `U_thread` contexts of its process and removes the actor implementing the process. Upon actor destruction, the CHORUS Nucleus frees all CHORUS entities associated with this actor.

4.5 Other UNIX Extensions

The CHORUS implementation of the UNIX subsystem has lead to several significant extensions which offer, at the UNIX subsystem level, access to CHORUS functionalities:

4.5.1 IPC

UNIX processes running on CHORUS can communicate with other UNIX processes, bare CHORUS actors, or entities from other subsystems using the CHORUS IPC mechanisms. In particular, processes are able to :

- create and manipulate CHORUS ports;
- send and receive messages;
- issue remote procedure calls.

4.5.2 Memory management

UNIX processes running on CHORUS can create, delete and share memory regions.

4.5.3 Real-Time

CHORUS real-time facilities provided by the Nucleus are available at the UNIX subsystem level to privileged applications:

- CHORUS provides the ability to dynamically connect handlers to hardware interrupts. This facility is already used by UNIX Device Managers.
- UNIX processes enjoy the benefit of the priority based preemptive scheduling provided by the CHORUS Nucleus.

Moreover, for interrupt processing, UNIX servers may immediately process an interrupt within the interrupt context or defer the majority of the processing to be handled by a dedicated thread executing

within the server context.

This functionality allows CHORUS/MiX device drivers to mask interrupts for shorter periods of time than they are masked in many standard UNIX implementations. Thus, with a little tuning, real-time applications can be made to run in a UNIX environment with better response time to external events.

5. CONCLUSION

CHORUS was designed with the intention of supporting fully-functional, industrial quality operating system environments. Thus, the inherent trade-off between performance and richness of the design was often made in favor of performance.

Making the CHORUS Nucleus facilities *generic* prevented the introduction of features with complex semantics. Features such as stringent security mechisms, application-oriented protocols, and fault tolerance strategies, do not appear in the CHORUS Nucleus. The CHORUS Nucleus provides, instead, the building blocks with which to construct these features inside subsystems.

CHORUS provides effective, high performance solutions to some of the issues known to cause difficulties to system designers:

- Exceptions are posted by the Nucleus to a port chosen by the actor program. This simple mechanism allows a user actor to apply its own strategy for handling exceptions and, because of the nature of ports, it extends transparently to distributed systems.

  Exceptions can also be associated directly with actor routines, for high performance within system actors.

- Debugging within CHORUS distributed systems is facilitated since resources are isolated within actors and since the message passing paradigm provides explicit and clear interactions between actors.

- The CHORUS modular structure allows binary compatibility with UNIX in CHORUS–V3, while maintaining a well structured, portable and efficient implementation.

The experience of four CHORUS versions has validated the CHORUS concepts. Unique Identifiers provide global, location independent names which form the basis for resource location within the a CHORUS distributed system. Actors and threads provide modular, high-performance, multi-threaded computational units. Messages, ports, and port groups provide the underlying communication mechanism with which CHORUS computational entities are bound together to construct distributed systems.

The CHORUS technology has the following features:

- it uses a communication-based architecture, relying on a minimal Nucleus which integrates distributed processing and communication at the lowest level, and which implements generic services used by a set of subsystem servers to extend standard operating system interfaces. A UNIX† subsystem has been developed; other subsystems such as OS/2 and object-oriented systems are planned;

- the real-time Nucleus provides real-time services which are accessible by system programmers;

- it is a modular architecture providing scalability, and allowing, in particular, dynamic configuration of the system and its applications over a wide range of hardware and network configurations, including parallel and multiprocessor systems.

The CHORUS technology has been designed to build a new generation of open, distributed, and scalable operating systems. In addition to providing the basis for the emulation of existing operating systems, CHORUS technology provides a means by which these subsystem interfaces can be extended transparently to exploit distributed environments. CHORUS technology can be used in conjunction with these subsystems, or independently, to assemble high-performance distributed applications.

6. ACKNOWLEDGEMENTS

Jean-Pierre Ansart, Allan Bricker, Philippe Brun, Hugo Coyote, Corinne Delorme, Jean-Jacques Germond, Steve Goldberg, Pierre Lebée, Frédéric Lung, Marc Maathuis, Denis Metral-Charvet, Douglas Orr, Bruno Pillard, Didier Poirot, Eric Pouyoul, François Saint-Lu and Eric Valette contributed, each with a particular skill, to the CHORUS-V3 implementation on various machine architectures.

Hubert Zimmermann by initiating the Chorus research project at INRIA, encouraging its development, and leading its transformation into an industrial venture made all this possible.

7. REFERENCES

[Abro89] Vadim Abrossimov, Marc Rozier, and Marc Shapiro, "Generic Virtual Memory Management for Operating System Kernels," in *Proc. of 12th. ACM Symposium on Operating Systems Principles*, Litchfield Park, AZ, (3-6 December 1989), 20 p. Chorus systèmes Technical Report CS/TR-89-18.2

[Acce86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, "Mach: A New Kernel Foundation for UNIX Development," in *USENIX Summer'86 Conference Proc.*, Atlanta, GA, (9-13 June 1986), pp. 93-112.

[Arma86] François Armand, Michel Gien, Marc Guillemont, and Pierre Léonard, "Towards a Distributed UNIX System – the CHORUS Approach," in *EUUG Autumn'86 Conference Proc.*, Manchester, UK, (22-24 September 1986), pp. 413-431.

[Arma89] François Armand, Michel Gien, Frédéric Herrmann, and Marc Rozier, "Revolution 89, or Distributing UNIX Brings it Back to its Original Virtues," in *Proc. of Workshop on Experiences with Building Distributed (and Multiprocessor) Systems*, Ft. Lauderdale, FL, (5-6 October 1989), pp. 153-174. Chorus systèmes Technical Report CS/TR-89-36.1

[Arma90] François Armand, Frédéric Herrmann, Jim Lipkis, and Marc Rozier, "Multi-threaded Processes in Chorus/MIX," in *Proc. of EUUG Spring'90 Conference*, Munich, Germany , (23-27 April 1990), pp. 1-13. Chorus systèmes Technical Report CS/TR-89-37.3

[Bani82] Jean-Serge Banino and Jean-Charles Fabre, "Distributed Coupled Actors: a CHORUS Proposal for Reliability," in *IEEE 3rd. International Conference on Distributed Computing Systems Proc.*, Fort-Lauderdale, FL, (18-22 October 1982), 7 p.

[Bani85] Jean-Serge Banino, Jean-Charles Fabre, Marc Guillemont, Gérard Morisset, and Marc Rozier, "Some Fault-Tolerant Aspects of the CHORUS Distributed System," in *IEEE 5th. International Conference on Distributed Computing Systems Proc.*, Denver, CO, (13-17 May 1985), pp. 430-437.

[Bét70] Claude Bétourné, Jacques Boulenger, Jacques Ferrié, Claude Kaiser, Sacha Krakowiak, and Jacques Mossière, "Process Management and Resource Sharing in the Multiaccess System ESOPE," *Communications of the ACM*, vol. 13, no. 12, (December 1970).

[Cher88] David Cheriton, "The Unified Management of Memory in the V Distributed System," Technical Report, Computer Science, Stanford University, Stanford, CA, (1988).

[Cher88a] David Cheriton, "The V Distributed System," *Communications of the ACM*, vol. 31, no. 3, (March 1988), pp. 314-333.

[Gien83] Michel Gien, "The SOL Operating System," in *Usenix Summer'83 Conference*, Toronto, ON, (July 1983), pp. 75-78.

[Ging87] Robert A. Gingell, Joseph P. Moran, and William A. Shannon, "Virtual Memory Architecture in SunOS ," in *Usenix Summer'87 Conference*, Phoenix, AR, (8-12 June 1987), pp. 81-94.

[Guil82] Marc Guillemont, "The CHORUS Distributed Operating System: Design and Implementation," in *ACM International Symposium on Local Computer Networks Proc.*, Florence, Italy, (April 1982), pp. 207-223.

[Guil91] Marc Guillemont, Jim Lipkis, Doug Orr, and Marc Rozier, "A second-Generation Microkernel-Based UNIX: Lessons in Performance and Compatibility," in *Usenix Winter'91 Conference*, Dallas, TX, (21-25 January 1991).

[Herr88] Frédéric Herrmann, François Armand, Marc Rozier, Michel Gien, Vadim Abrossimov, Ivan Boule, Marc Guillemont, Pierre Léonard, Sylvain Langlois, and Will Neuhauser,

"CHORUS, a New Technology for Building UNIX Systems," , Cascais, Portugal, (3-7 October 1988).

[Lega88] José Legatheaux-Martins and Yolande Berbers, "La Désignation dans les Systèmes d'Exploitation Répartis," *Technique et Science Informatique*, vol. 7, no. 4, (Juillet 1988), pp. 359-372.

[Li86] Kai Li, "Shared Virtual Memory on Loosely Coupled Multiprocessors," PhD. Thesis, Yale University, New-Haven, CT, (September 1986).

[Mora88] Joseph P. Moran, "SunOS Virtual Memory Implementation," in *EUUG Spring'88 Conference*, London, UK, (11-15 April 1988), pp. 285-300.

[Mull87] Sape J. Mullender et al., *The Amoeba Distributed Operating System: Selected Papers 1984 -1987*, CWI Tract No. 41, Amsterdam, Netherlands, (1987), 309 p.

[Nels88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout, "Caching in the Sprite Network File System," *ACM Transactions on Computer Systems*, vol. 6, no. 1, (February 1988), pp. 134-154.

[Pouz82] Louis Pouzin et al., *The CYCLADES Computer Network - Towards Layered Network Architectures*, Monograph Series of the ICCC, 2, Elsevier Publishing Company, Inc, New-York, NY, (1982), 387 p. ISBN 0-444-86482-2

[Pres86] David L. Presotto, "The Eight Edition UNIX Connection Service," in *EUUG Spring'86 Conference Proc.*, Florence, Italy, (21-24 April 1986), 10 p.

[Rash87] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," in *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, (October 1987), pp. 31-39.

[Rozi87] Marc Rozier and José Legatheaux-Martins, "The CHORUS Distributed Operating System: Some Design Issues," in *Distributed Operating Systems, Theory and Practice*, Yakup Paker, Jean-Pierre Banâtre and Muslim Bozyigit ed., NATO ASI Series, vol. F28, Springer Verlag, Berlin, (1987), pp. 261-287.

[Tane86] Andrew S. Tanenbaum, Sape J. Mullender, and Robert van Renesse, "Using Sparse Capabilities in a Distributed Operating System," in *IEEE 6th. International Conference on Distributed Computing Systems*, CWI Tract No. 41, Cambridge, MA, (19-23 May 1986), pp. 558-563.

[Wein86] Peter J. Weinberger, "The Eight Edition Remote Filesystem," in *EUUG Spring'86 Conference*, Florence, Italy, (21-24 April 1986), 1 p.

[Zimm81] Hubert Zimmermann, Jean-Serge Banino, Alain Caristan, Marc Guillemont, and Gérard Morisset, "Basic Concepts for the Support of Distributed Systems: the CHORUS Approach," in *IEEE 2nd. International Conference on Distributed Computing Systems Proc.*, Versailles, France, (April 1981), pp. 60-66.

[Zimm84] Hubert Zimmermann, Marc Guillemont, Gérard Morisset, and Jean-Serge Banino, "CHORUS: a Communication and Processing Architecture for Distributed Systems," Research Report, INRIA, Rocquencourt, France, (September 1984).

Modularity and Interfaces in Micro-Kernel Design and Implementation: A Case Study of Chorus on the HP PA-RISC*

Jonathan Walpole, Jon Inouye, and Ravindranath Konuru
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
(walpole,jinouye,konuru@cse.ogi.edu)

ABSTRACT

The key concept that distinguishes micro-kernel operating systems from their macro-kernel counterparts is modularity. Micro-kernels implement operating system functionality in well-defined modules with clearly identified interfaces between them. Proponents of this modular approach to operating system design claim that it offers advantages in the areas of portability, correctness, protection, extensibility, and reconfigurability for distributed architectures. If micro-kernels are to gain wider acceptance however, it is important to ensure that these benefits of modularity can be attained without incurring significant performance degradation when compared to macro-kernels.

In this paper we explore the relationship between modularity and performance by examining an implementation of the Chorus micro-kernel operating system on the Hewlett-Packard PA-RISC workstation. We outline the key interfaces in Chorus and study the architectural assumptions implicit in these interfaces.

1 Introduction

The key characteristic that distinguishes micro-kernel operating systems from their macro-kernel counter parts is modularity. Micro-kernel operating systems are structured as a collection of cooperating servers running above a minimal kernel. Structuring operating systems in this manner offers a number of potential benefits including ease of distribution, reconfigurability, extensibility, portability, protection and correctness [4, 5].

It has been argued that ease of distribution and reconfigurability result from the separation of system components and the use of message passing as the communication mechanism among them. Similarly, this separation of system components is claimed to improve extensibility by allowing new operating system functionality to be added, in the form of new system servers, without altering existing components or the micro-kernel itself. Arguments about improved correctness are based on the principle that it is easier to avoid design and programming errors if a system is composed from several small modules rather than a single large module. Modular systems also allow protection to be enforced at the boundaries between modules. Finally, it is claimed that micro-kernel operating systems improve portability by localizing machine-dependent code within the micro-kernel.

*This research is supported by the Hewlett-Packard Company, Chorus Systèmes, and the Oregon Advanced Computing Institute (OACIS).

A central research issue in the construction of real-world micro-kernel operating systems is the design of interfaces that allow the above benefits to be attained while achieving performance comparable to less modular, macro-kernel operating systems. This is a major challenge for micro-kernel designers because macro-kernels implement very low-overhead invocation across their internal interfaces by using local procedure calls. In order to achieve all the proposed benefits of modularity, micro-kernels must support a variety of interfaces and invocation mechanisms, many of which are considerably more complex and heavy weight than a local procedure call.

The challenge for micro-kernel designers is to (a) define interfaces that are expressive enough to allow the above benefits to be attained, and (b) to provide implementations of those interfaces that offer acceptable performance. Unfortunately, in making these implementation choices operating system designers are often forced to trade the sought after benefits of modularity for performance. The degree to which this occurs depends on two main factors. First, the anticipated requirements of the target application domain determine the relative importance of modularity and performance. For example, some target application domains may consider runtime protection and dynamic reconfiguration to be critical, and worth sacrificing performance for, whereas others may consider performance to be of paramount importance.

Second, the characteristics of the architecture, or class of architectures, on which the operating system is expected to execute may determine the efficiency with which particular functionality can be supported. Making such implementation decisions based on key architectural assumptions is already common practice in operating system design. However, the success of this approach depends heavily on the accuracy of these assumptions. Consequently, operating system designers must be aware of the trends in computer architecture (and vice versa). This issue is important both for the implementation of efficient interfaces for system structuring, and for defining an appropriate separation between portable and non-portable code.

We believe that the definition of appropriate interfaces, and the implementation decisions that determine the balance between performance and modularity will be critical to the eventual success or failure of micro-kernel operating systems. In this paper we explore this design space by studying our implementation of the Chorus micro-kernel on the Hewlett-Packard Precision Architecture RISC (PA-RISC) 9000/834 workstation.

The remainder of the paper is organized as follows. Section 2 outlines the Chorus approach to modularity and discusses the interfaces defined by Chorus. The trade-off between performance and modularity in the implementation of those interfaces, and the architectural assumptions implicit in the implementation choices, are also discussed. Section 3 outlines the key characteristics of the PA-RISC, discusses the salient features of our implementation of Chorus, and revisits the architectural assumptions implicit in Chorus. Section 4 summarizes the strengths and weaknesses of the Chorus approach, and section 5 concludes the paper.

2 Modularity and Interfaces in Chorus

A Chorus-based operating system is structured as a set of cooperating subsystem servers executing above a Chorus nucleus (henceforth called the micro-kernel) [2]. There are a number of key interfaces in this operating system structure (see figure 1). A high-

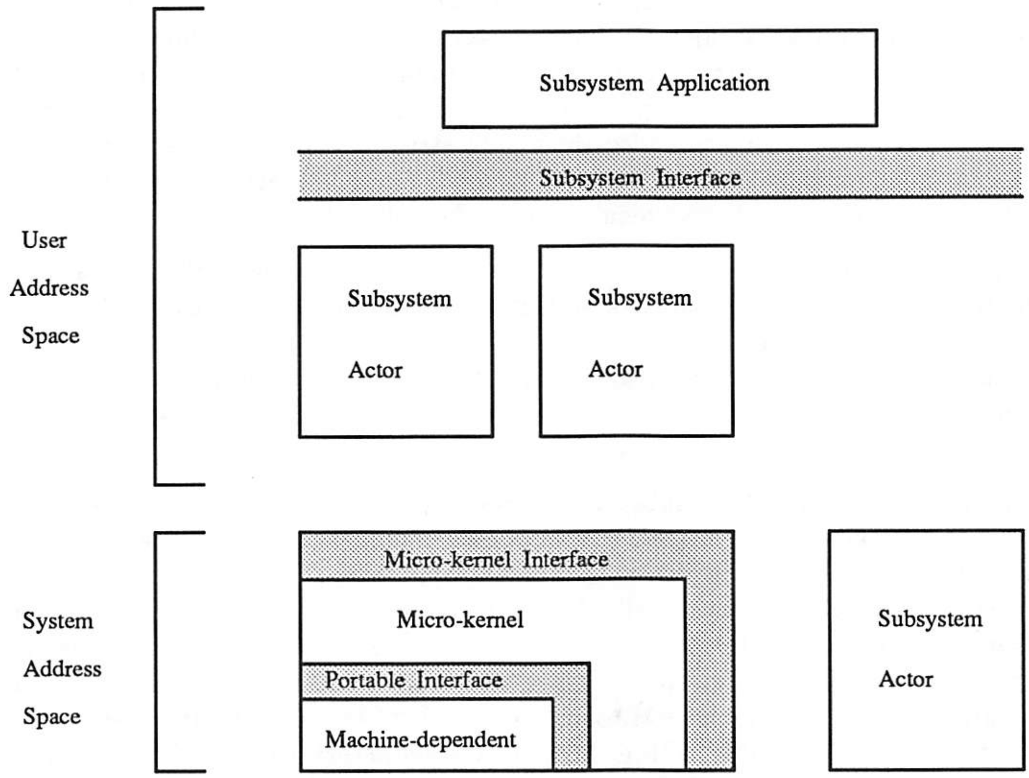


Figure 1: A typical Chorus Operating System Structure

level operating system interface, such as the UNIX system call interface, is presented by the subsystem servers to application programs. We will refer to this as the *subsystem interface.* A lower-level interface, called the *micro-kernel interface* defines the interaction between the micro-kernel and the subsystem servers. The micro-kernel interface exports a number of basic abstractions including IPC which is used to build higher-level interfaces between subsystem servers. An additional interface, defined within the micro-kernel, separates machine-dependent code from portable code. We will refer to this as the *portable interface.* The portable interface is intended to be architecture-independent, and is supported by a new implementation of the micro-kernel's machine-dependent layer for each new architecture on which it executes.

Some of the basic abstractions exported by the micro-kernel interface are ports, messages, threads and actors[1]. Chorus defines two types of actor (*user* and *supervisor*) based on their privilege level and allowed operations [10]. User actors are placed in separate address spaces and system actors share the system address space. In addition to defining operations for the creation and manipulation of the basic micro-kernel abstractions, the micro-kernel interface allows subsystem servers to attach handlers to traps, interrupts, and exceptions.

User and supervisor actors see the same specification of the micro-kernel interface, but use distinct implementations of the interface. These implementations take the form of separate libraries above the interface, and separate vectors of routines below the interface. The use of a common interface specification allows the decision of whether to load an actor into system or user space to be delayed until link time.

[1] An actor is the unit of resource allocation in Chorus, similar to a task in Mach [1].

During a micro-kernel interface call, stubs in the library for user actors load the call number into a temporary register and use an architecture-specific instruction to cause a change in privilege level[2]. In contrast, the stubs in the library for supervisor actors take advantage of the fact that supervisor actors are in the same address space as the micro-kernel by using a global data structure, called the ROOT structure, to locate the appropriate vector of routines in the micro-kernel. The call stub for a supervisor actor thus reduces to little more than a procedure call to the required micro-kernel routine.

The supervisor actor concept offers several important advantages. First, it allows heavily used IPC paths between subsystem servers to be streamlined by using IPC calls in the optimized implementation of the micro-kernel interface. Second, it allows trap-based subsystem interfaces to be implemented efficiently by directly calling subsystem call handlers rather than passing control back up to an emulation library in the application's address space.

For example, the Chorus MiX subsystem[3] presents a UNIX compatible subsystem interface by using a supervisor actor (called the process manager (PM)) to attach system call handlers to UNIX-specific trap numbers. When an application makes a MiX system call a trap is generated and the micro-kernel calls the handler attached by the PM. Since the PM is a supervisor actor and runs in the system address space it can access the call parameters directly in the application's address space. Once the system call has been identified, it is either executed directly within the PM, or passed via IPC to another subsystem actor (such as the object manager[4] (OM)). Other MiX subsystem actors also attach handlers to hardware events. For example, the OM connects handlers to disk interrupts, i.e., the disk driver is part of the OM.

2.1 Architectural Assumptions in Chorus

In all of its interfaces, Chorus is fairly successful in achieving the benefits of modularity without sacrificing too much performance. This is largely a result of good design decisions relating to the separation of specification from implementation in its interfaces. However, there are several key assumptions implicit in the design decisions that led to the current interface definitions. These assumptions are aimed at optimizing performance for some "common class" of architectures.

First, the decision to load supervisor actors into the system address space is based, in part, on the assumption that invocation between user-level entities is considerably more expensive than invocation between entities residing in the system address space. Second, because some supervisor actors need to perform privileged instructions, to access hardware for example, Chorus places them in the system address space. The basic assumption here is that an actor can be privileged only if it runs in the system address space.

The placement of supervisor actors in the system address space in order to support the attachment of handlers to traps and interrupts involves similar assumptions, i.e., that efficient handler invocation requires the handler to reside in the system address space. This kind of assumption is related to the relative cost of cross-address space and intra-address

[2] On most architectures this is a trap-based instruction.

[3] MiX is a UNIX System V compatible subsystem that runs above the Chorus micro-kernel. We had access to Chorus MiX 3.2.

[4] The object manager implements file service and acts as a default mapper.

space communication and is implicit in the decision to include the disk driver in the MiX 3.2 object manager.

Finally, the portable interface specification involves a number of architectural assumptions. Of specific interest in our experiments with Chorus is the extensive use of memory mapping in and above the portable interface. This assumes that memory mapping is inexpensive, as is the case on architectures with physically addressed caches.

The design and implementation decisions discussed above are not unusual. In fact, they are based on "common knowledge" about traditional computer architecture. In the following sections we argue that the characteristics of current-generation architectures have begun to change, and that operating system designers may need to re-evaluate some of these basic architectural assumptions.

3 Supporting the Chorus Interfaces on the PA-RISC

The PA-RISC is the framework for HP's 3000/900, 9000/800, and 9000/700 series computer systems. During 1991 we ported the Chorus v3.3 micro-kernel to the Hewlett-Packard 9000/834 workstation, and studied the interaction between Chorus and the PA-RISC architecture. This section discusses the issues involved in implementing the Chorus interfaces on the PA-RISC and revisits the architectural assumptions implicit in those interfaces.

The PA-RISC provides a 64-bit global address space that is shared between all processes and the operating system. Virtual memory is partitioned into segments, called *address spaces*, each containing 2^{32} bytes. Our implementation of Chorus uses the first of these 32-bit address spaces for the system address space which is further divided into partitions for the micro-kernel, supervisor actors, and the ROOT structure. Virtual addresses consist of two components: a space identifier and a space offset. In short pointer (32-bit) addressing mode the space identifier is held in a space register which is identified using the two most significant bits of the 32-bit offset. In long pointer (64-bit) addressing mode both parts of the address are specified explicitly. The PA-RISC also provides specific instructions for intra-space and inter-space branching.

Protection between processes sharing the global virtual address space is supported using protection identifiers, access identifiers, and access rights. Protection identifiers are associated with processes, whereas access identifiers and access rights are associated with virtual memory pages. Access rights specify the types of access that are allowed at different privilege levels, and access identifiers are used in combination with protection identifiers to implement capability-style protection. During execution, four protection registers are available to store some of the protection identifiers associated with a process. In order to make a successful access, the requested operation and privilege level must pass the access rights check, and one of the four protection registers must contain a protection identifier that matches the page's access identifier[5].

In our Chorus implementation we use a special access identifier (0) for the code and data pages of the micro-kernel and supervisor actors. This has special significance on the PA-RISC, in that it matches all protection identifiers. Access rights are used to prevent accesses by code running at user privilege level. However, since the micro-kernel and supervisor

[5]It is possible to associate more than four protection identifiers with a process if a handler is provided to manage the protection fault generated during a match failure at access time.

actors all run at the highest privilege level this does not prevent them from accessing each others' pages.

In common with other architectures, promotion of privilege level can be triggered during a trap or hardware exception. However, the PA-RISC also provides more efficient support for privilege promotion and reduction. Privilege promotion can occur, without causing a trap, by executing a *gate* instruction on a specially protected page, called the *gateway page* which is mapped, execute-only, into a pre-defined location in the system address space. In our implementation, the access rights on the gateway page are set such that the execution of a gate instruction will cause promotion to the highest privilege level.

Efficient privilege reduction is supported using a two-level instruction address queue which supports delayed branching. The two least significant bits of the instruction address are not needed since instructions lie on 4-byte boundaries. These bits are used instead to keep track of the current execution privilege, and can be set during a branch instruction in order to *reduce* the current privilege level. This feature is used during the return from a system call.

3.1 The Micro-Kernel Interface

The implementation of the micro-kernel interface for user actors makes use of the above features in the following manner:

- A thread in a user actor makes a micro-kernel interface call by executing a stub in the micro-kernel interface library for user actors (**chorusLib.a**).
- The call stub loads a subsystem number and a call number into temporary registers. This allows the micro-kernel to marshal calls intended for a subsystem interface rather than the micro-kernel interface. In order to distinguish micro-kernel interface calls, the micro-kernel is assigned a special subsystem number.
- The call stub then executes an inter-space branch instruction to the gateway page in the system address space.
- A gate instruction is executed in the gateway page, which causes privilege promotion, and an intra-space branch instruction is executed into the appropriate entry point in the system address space.
- The micro-kernel switches to the system stack and saves the necessary registers. If the subsystem number indicates a micro-kernel interface call the micro-kernel copies the parameters from the user stack and calls the appropriate routine in the call vector for user actors. Otherwise, a handler attached by a specified subsystem actor is called. The subsystem actor is then responsible for copying parameters from the user stack and performing the subsystem-specific call.
- After the call has been serviced by the micro-kernel, or by a subsystem actor, the micro-kernel completes the call by switching stacks and using the delayed branch instruction to return to user space and re-establish the original privilege level. Note that no trap was necessary to handle the system call.

The implementation of the micro-kernel interface for supervisor actors differs from that for user actors in the following way:

- A thread in a supervisor actor makes a micro-kernel interface call by executing a stub in the micro-kernel interface library for supervisor actors (**chorusSvLib.a**).

- The call stub loads the address of the micro-kernel's call vector for supervisor actors from the ROOT structure[6]. The stub uses the call number to calculate the address of the appropriate routine in the call vector for supervisor actors and uses it to call the routine. This does not require a change of stacks because supervisor threads always execute on a system stack.

Note that this implementation of the micro-kernel interface is only an optimization rather than a change in the interface specification. Supervisor actors may still use the user actor micro-kernel interface library without a loss in functionality.

3.2 The Portable Interface

In addition to affecting the implementation of the micro-kernel interface, the PA-RISC's architectural features also had a significant impact on the implementation of the portable interface. In fact, the majority of the work involved in our port of Chorus to the PA-RISC was associated with building a new implementation of the portable interface. There were a number of interesting aspects to this work, particularly in the area of cache management. The PA-RISC uses a virtually addressed cache to improve performance by allowing TLB look-up and cache access to be performed in parallel [9]. As with other architectures that use virtually addressed caches (such as the IBM RS-6000 and the MIPS 4000) the PA-RISC relies on the operating system to maintain *address translation consistency*. In other words, address aliases falling in different cache sets must be resolved by the operating system. Such aliases are generally created by mapping multiple virtual addresses to the same physical address. However, since the PA-RISC also uses the cache when executing in physical addressing mode, aliases can arise when the cache index generated by a physical address differs from the index produced when accessing the same data using a currently mapped virtual address. In either case, the operating system must prevent any loss of consistency due to the concurrent placement of the same data item in different cache lines.

Unfortunately, the Chorus portable interface contains several functions that can generate aliases, and the portable layers of Chorus assume that such aliases are inexpensive (both to set up and maintain). In order to support aliasing at the portable interface, the machine-dependent layers must maintain address translation consistency. In our implementation we achieved this by using a technique called *pseudo-aliasing*. Pseudo-aliasing guarantees that for any physical page, only one mapping can exist in the cache and TLB at a time. When an access is attempted via a virtual address that is logically valid, but for which the mapping has been removed from the cache and TLB, a *pseudo page fault* occurs. During pseudo page fault handling any existing mapping to the physical page is invalidated, the cache lines and TLB entry associated with it are flushed, and a new mapping is established. This approach presents a semantically correct implementation of the portable interface. However, it significantly changes the relative costs of certain primitive virtual memory operations. In particular, maintaining multiple mappings to the same page, and unmapping a page, become expensive (see [7] for more details).

[6] This is set up during kernel initialization.

3.3 Revisiting the Architectural Assumptions in Chorus

Our initial implementation of Chorus did not take full advantage of many of the more interesting features of the PA-RISC such as the global address space, protection, and privilege manipulation facilities. This was largely due to caution: we wanted to avoid making any radical departures from the standard Chorus approach in our first implementation. Consequently, we implemented Chorus as if the architectural assumptions discussed in section 2 held for the PA-RISC. This section revisits those assumptions and discusses alternative approaches to implementing Chorus on the PA-RISC (see [11, 6, 8, 12] for a more detailed discussion of our implementation and possible alternative approaches).

First, Chorus tends not to distinguish between issues of address space, protection domain and privilege level. Actors either run as supervisor actors at the same privilege level, and in the same address space and protection domain as the micro-kernel, or they run as user actors at the lowest privilege level, and in their own address space and protection domain. On the PA-RISC the concepts of address space, protection domain and privilege level are orthogonal. Therefore, it is feasible to use arbitrary combinations of these features in the implementation of different types of actors. This approach leads to a range of possible invocation costs depending on the specific boundaries between actors.

In order to gain a better understanding of these costs we profiled our Chorus implementation. Table 1 illustrates costs for a cross address space call at the same privilege level and protection domain, and null system calls using both the user and supervisor implementations of the micro-kernel interface[7].

For comparison, the cost of a null procedure call is also presented. Contrary to Chorus' assumptions, these figures show that the cost of an inter-space, intra-protection domain call (3.4 μ seconds) is not dramatically higher than the cost of an intra-space, intra-protection domain call (0.9 μ seconds). This suggests that, on the PA-RISC, groups of actors which communicate heavily do not necessarily have to be placed in the same address space in order to exhibit good performance.

Table 1: Basic invocation costs

Call-type	Time (in μ sec)
null procedure call	0.9
supervisor system call	2.2
user-mode system call	12.3
cross address space	3.4

Chorus also places supervisor actors in the system address space to optimize the performance of invocations across the micro-kernel interface. Chorus assumes that micro-kernel entry and exit costs from user space are high relative to the cost of the supervisor implementation of the micro-kernel interface. Our figures for the PA-RISC show that micro-kernel calls from supervisor actors are about one fifth the cost of micro-kernel calls from user actors.

[7]The 9000/834 provides a timer with a resolution of 1/15 of a μ second.

Table 2: A breakdown of user-mode system call costs

Stage	Time (in μ sec)
kernel entry	1.6
kernel processing	10.5
kernel exit	0.2

A further breakdown of the component costs involved in a user-mode system call is presented in table 2. The cost of promoting privilege level via the gateway page, followed by a branch into the system address space is illustrated in the kernel entry figure (1.6 μ seconds). The kernel exit figure (0.2 μ seconds) illustrates the cost of privilege reduction using the delayed branch instruction. The remaining time (10.5 μ seconds) is taken in kernel processing which includes saving and restoring registers[8], switching stacks, and making the system call.

The difference in cost between a user-mode system call and a cross address space call is due to the cost of changing privilege level and protection domain [3]. Since the cost of changing privilege level using the gateway page is known to be around 1.6 μ seconds, we can deduce that the remainder of the cost is associated with the protection domain transfer. Further investigation of this costs shows the dominant cost in protection domain transfer to be associated with switching execution stacks. Consequently, the difference in cost between supervisor and user calls across the micro-kernel interface is due mainly to the use of a protected call rather than the use of different privilege levels or address spaces.

Chorus also assumes that the execution of privileged instructions requires a supervisor actor to be in the system address space. Since address space and privilege level are orthogonal issues on the PA-RISC, it is possible to place supervisor actors outside the system address space and still allow them to execute privileged instructions. Such actors need not execute continually in privileged mode since the gateway mechanism allows changes in privilege level that are considerably more efficient than trap-based mechanisms. On the PA-RISC the cost of increasing privilege level via the gateway page is around twice the cost of a null procedure call (i.e., 1.6 μ seconds) and the cost of lowering privilege is only around 0.2 μ seconds.

A related assumption is that an actor must be in the system address space to handle hardware events efficiently. This is based on the principle that the identification of a virtual address for a handler in another address space requires significant overhead for memory context set-up. The PA-RISC's global address space ensures that every virtual address can be uniquely identified. Furthermore, the cost of an inter-space call (3.4 μ seconds) is not dramatically higher than the cost of a null procedure call (0.9 μ seconds). Thus, supervisor actors need not reside in any specific 32-bit address space in order to handle hardware events.

Consequently, there are a number of feasible options for implementing supervisor actors on the PA-RISC. For example, they could be given their own private 32-bit address

[8]Our measurements of user system call cost do not include the cost of saving and restoring co-processor context.

space, or they could reside in the system address space. If they reside in the system address space they could be given distinct protection and access identifiers, or they could use the same protection domain and privilege level as the kernel. Each of these possible variants would exhibit slightly different characteristics with respect to modularity and performance, and each would require a slightly different implementation of the micro-kernel interface.

The specification of the Chorus portable interface also involves some assumptions that are inappropriate for the PA-RISC. In particular, software maintenance of address translation consistency causes the relative costs of certain primitive memory management operations to alter significantly. For example, cache flushing during an unmapping operation for a 2 K-byte page can increase the cost of the operation by between 150% and 1000% depending on the state of the cache [7]. Changes in the cost of primitive operations supported in the portable interface can, in turn, lead to inappropriate design decisions in higher-level, portable code. A classic example is the use of memory mapping, rather than copying, to implement IPC. On architectures with a physically addressed cache, mapping is a clear win over byte-copying for page-sized, page-aligned data. However, the expense of cache flushing on a virtually addressed cache architecture can decrease the performance of IPC implementations based on mapping to the extent that, under certain circumstances, byte copying implementations can be as fast as, or even faster than, mapping implementations (For an in-depth discussion of the effects of virtually addressed caches on virtual memory design and performance, and for more details of the performance results discussed here, see [7]).

A better long term solution would be to make use of the PA-RISC's global address space and long pointer addressing in order to avoid address aliasing problems. This would require a major redesign of Chorus to remove the notions of private per-process address spaces, and to share memory only via the global address space using unique 64-bit virtual addresses rather than by mapping multiple virtual addresses to the same physical page.

4 Strengths and Weaknesses of the Chorus Approach

One of the major strengths of the Chorus approach is its separation of interface specification from implementation. The interfaces between operating system components are specified in terms of IPC which is supported in the specification of the micro-kernel interface. An interesting feature of Chorus is its use of two distinct implementations of the micro-kernel interface: the supervisor actor interface and the user actor interface. Each implementation makes a different trade-off between modularity and performance. The supervisor actor interface is implemented in a manner that preserves dynamic reconfigurability, but trades runtime protection for performance. User actors, on the other hand, maintain protection at the expense of performance by using a more heavy weight invocation mechanism when crossing the micro-kernel interface.

The provision of these two separate implementations of what is essentially the same interface allows Chorus to support multiple different implementations of the same modular operating system. The main problem, however, is that there are only two choices: a system component must be either a supervisor actor, which uses an efficient but unprotected implementation of the micro-kernel interface, or a user actor which uses a protected but relatively slow implementation of the interface. For some application domains neither of these choices may be appropriate. Similarly, the loss of modularity in exchange for performance, which is implicit in these choices, may be unnecessary for some architectures, particularly if the

architectural assumptions on which the choices are based are inappropriate.

It would be useful to generalize the approach taken in Chorus by allowing arbitrarily many implementations of a single interface specification. This would allow operating systems to be customized for different architectures and application domains. In order to support this level of customization it must be possible to localize the code representing a particular implementation of an interface. Above the interface this code is generally localized in a system call library. For example, Chorus provides two separate micro-kernel interface libraries, one for linking with supervisor actors and the other for user actors. Below the interface, the code associated with a particular implementation must also be localized if customization is to be supported efficiently. In Chorus such code is localized in system call vector modules: one for handling calls originating in the supervisor actor implementation of the micro-kernel interface, and the other for user actor calls.

Another strength of Chorus is its separation of portable and machine-dependent code. Chorus defines a portable interface, internal to the micro-kernel, that is implemented using machine-dependent code. However, this interface is so close to the underlying hardware that it is difficult to avoid building architectural assumptions into the interface specification. This can have the unfortunate effect of fostering inappropriate assumptions in higher-level code that uses the portable interface. Our experience with Chorus on the PA-RISC indicates that even though we are able to support a semantically correct version of the portable interface, the relative performance of some of the operations within it put into question various design decisions made in the portable layers.

5 Conclusion

This paper has discussed the relationship between modularity and interface design in micro-kernel operating systems. Modularity is a major potential strength of micro-kernel based systems. However, in order to gain wider acceptance, micro-kernels must exhibit performance comparable to monolithic operating systems. An important principle in achieving this goal is the separation of interface specification from implementation. Interface implementation decisions are then largely concerned with establishing an appropriate balance between modularity and performance. Since this balance varies for different application domains and computer architectures, the most promising approach for micro-kernel designers is to offer customization by allowing multiple different implementations of the same interface specification. Chorus has begun to apply this principle by offering two different implementations of its micro-kernel interface. We believe that this approach should be generalized.

This paper has also outlined some of the architectural assumptions that are implicit in the design and implementation of operating system interfaces. Some of these assumptions, particularly those relating to cache design, are becoming out-dated given recent trends in computer architecture, and can lead to a relative decline in operating system performance.

6 Acknowledgements

Many people participated in the port of Chorus to the PA-RISC. We are particularly grateful to Marion Hakanson of OGI; Bart Sears of Hewlett-Packard Laboratories; Pascal Dietrich and Philippe Voisin of the University of Nancy; Vadim Abrossimov, Jean-Jacques Germond, Frédéric Herrmann, Olivier Giffard, and Marc Rozier of Chorus Systèmes.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation For UNIX Development. In *Proceedings of the 1986 Summer USENIX Conference*, pages 93–112, Atlanta, Georgia, 1986.

[2] François Armand, Michel Gien, Frédéric Herrmann, and Marc Rozier. Revolution 89 or "Distributing UNIX Brings it Back to its Original Virtues". In *Proceedings of the Workshop on Experiences with Building Distributed and Multiprocessor Systems*, October 5-6 1989.

[3] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight Remote Procedure Call. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 102–113, December 3-6 1989.

[4] Michel Gien. Micro-Kernel Design. *UNIX REVIEW*, 8(11):58–63, November 1990.

[5] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. Unix as an Application Program. In *Proceedings of the 1986 Summer USENIX Conference*, pages 87–95, Anaheim, California, 1990.

[6] Jon Inouye, Marion Hakanson, Ravindranath Konuru, and Jonathan Walpole. Porting Chorus to the PA-RISC: Virtual Memory Manager. Technical Report CSE-92-005, Oregon Graduate Institute, January 1992.

[7] Jon Inouye, Ravindranath Konuru, Jonathan Walpole, and Bart Sears. The Effects of Virtually Addressed Caches on Virtual Memory Design & Performance. Technical Report CSE-92-010, Oregon Graduate Institute, March 1992.

[8] Ravindranath Konuru, Marion Hakanson, Jon Inouye, and Jonathan Walpole. Porting the Chorus Supervisor and Related Low-Level Functions to the PA-RISC. Technical Report CSE-92-006, Oregon Graduate Institute, January 1992.

[9] Ruby B. Lee. Precision Architecture. *IEEE Computer*, 22(1):78–91, January 1989.

[10] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrman, Claude Kaiser, Sylvain Langlois, Pierre Léonard, and Will Neuhauser. Chorus Distributed Operating Systems. *Computing Systems Journal*, 1(4):305–370, December 1988.

[11] Jonathan Walpole, Marion Hakanson, Jon Inouye, and Ravindranath Konuru. Porting Chorus to the PA-RISC: Project Overview. Technical Report CSE-92-003, Oregon Graduate Institute, January 1992.

[12] Jonathan Walpole, Marion Hakanson, Jon Inouye, and Ravindranath Konuru. Porting Chorus to the PA-RISC: Overall Evaluation. Technical Report CSE-92-008, Oregon Graduate Institute, January 1992.

A Micro Kernel Architecture for Next Generation Processors

Toshio OKAMOTO * *Hideo SEGAWA* *Sung Ho SHIN*
Hiroshi NOZUE *Ken-ichi MAEDA* *Mitsuo SAITO*

Information Systems Lab.
Research and Development Center
TOSHIBA Corporation

April 27, 1992

Abstract

The authors made the best use of the huge address space provided by a 64-bit next generation architecture. A new micro kernel was designed with two outstanding features; *single virtual space* and *one level storage.* Three major benefits from the proposed kernel are *fast context switching*, *fast function call*, and *fast data access.*

This micro kernel manages only two abstractions to simplify the concept; *the thread* and *the memory section.*

The problem regarding how to prevent access by unauthorized threads or programs which first occurred due to the single virtual space has been finally solved with newly designed rich-functioned MMU hardware.

1 Introduction

This paper presents a new micro kernel architecture for next generation 64-bit processors, such as MIPS R4000 (Mashey[1]) and DEC Alpha. It has been found through discussions that ordinary ports for conventional OSs, such as UNIX or Mach (Rashid et al. [2]), cannot make the best use of the 64-bit address space for the following reasons.

- Conventional OS architectures have been designed under 16-bit or 32-bit limited address space constraints. A virtual space is allocated for a process. Context switching is therefore costly.

*1, Komukai Toshiba-cho, Saiwai-ku, Kawasaki, 210, JAPAN (email: oka@isl.rdc.toshiba.co.jp)

- Distributed environment development has resulted in the popularization of a client-server programming style, especially in the workstation environment. High speed interprocess communication (IPC) is required for executing such a program. An efficient user-level IPC using a shared memory to avoid kernel-level message passing overhead becomes central to the design of contemporary operating systems (Bershad et al. [3]).
- Usually, 64-bit address space is unnecessary except for special scientific calculations which process size exceeds the 32-bit address space (Mashey[1]).

This paper describes the design of the authors' new micro kernel architecture with a new memory model corresponding to a single 64-bit address space, which aims at simpler and better performance for shared services and multimedia applications, in particular on a workstation.

- *Single virtual storage (SVS)* enables a remote procedure call (RPC) to be executed like a subroutine call without changing the address spaces between a client program and a server program.
- *One-level storage (OLS)* addresses all the storages and manages the main memory in a one-level integrating a file region and a process region.
- *A fine-grained memory protection mechanism* within the SVS, using a rich-functioned memory management unit (MMU), prevents unauthorized access.

Besides the distinctive memory model feature, the new micro kernel introduces several achievements in the contemporary distributed operating system, such as

- UNIX and Mach binary compatibility
- Network transparency
- Real time execution

2 Micro Kernel Concept

2.1 Abstractions

This micro kernel handles only two kinds of resource abstractions.

- A *thread* is the basic execution unit and represents a flow for control or a register set.
- A *memory section* is a unit in the SVS with its backing storage. It holds certain attributes with respect to memory protection. A physical memory works as a backing storage cache.

The abstractions mostly follow those for Mach, but differ in the following points.

- A Mach thread can access within the address space for a specified task, whereas this thread can migrate from one procedure to another without switching the address space.
- The memory section resembles Mach memory object realization on the OLS. The memory sections are addressed uniquely, because of the SVS, whereas Mach comprises multiple virtual spaces each of which can be addressed independently.

Mach	New Micro Kernel
thread	thread
task	memory section with ACL
port	address on SVS
message	subroutine call on SVS
memory object	content of memory section

Table 1: Abstractions Compared with Mach

- This micro kernel will not handle a message passing. A message server emulates Mach message passing functions.
- This micro kernel will not handle a port either. The address on SVS with an access control list (ACL) corresponds to the Mach port mechanism. A port server supports the Mach port functions.

Table 1 shows the abstraction correspondence to Mach.

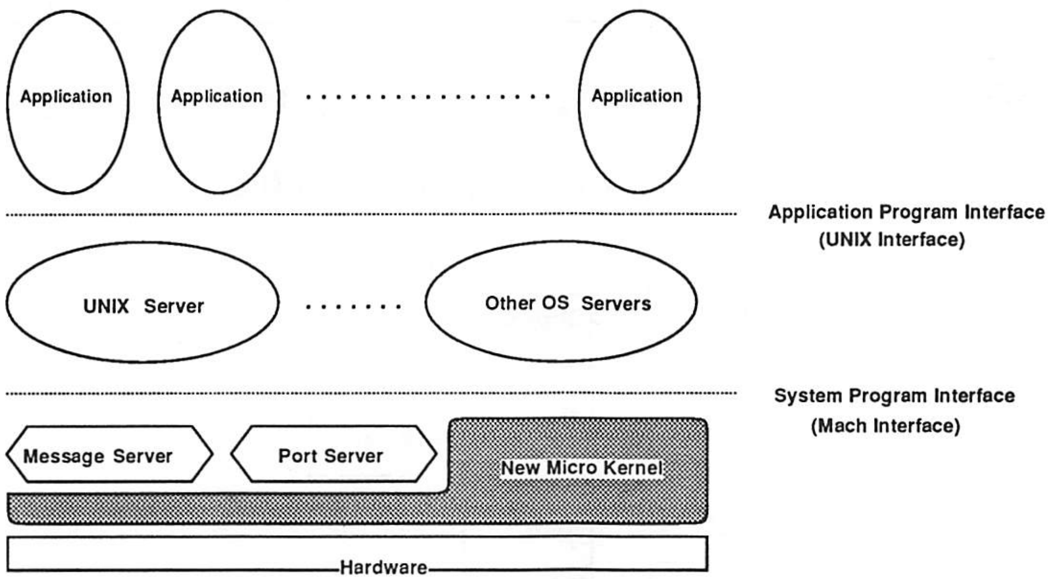


Figure 1: New Micro Kernel Structure

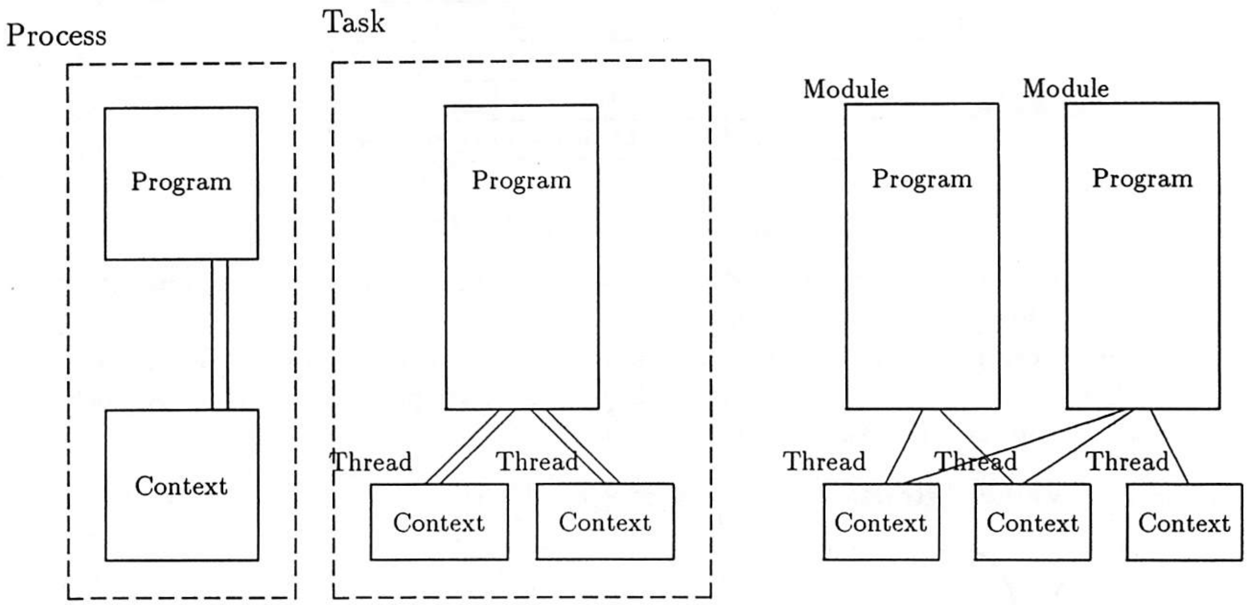
2.2 Programming Model

The UNIX programming model is based on a *process* which has one thread and one task in the Mach term. Mach splits the UNIX process into two elements, a task and a thread. Mach allows multiple threads to run within the context of exactly one task. This model is suited to the parallel execution of the same program by multiprocessors.

The new programming model is an extension of that proposed by ToM (Hagino et al. [4]) [1] for SVS. ToM programming model differs from Mach in that a thread

[1] ToM is a distributed operating system developed by the ToM Consortium, comprising Kyoto Univ., Keio Univ., ASTEM RI/Kyoto, and several Japanese corporations.

can migrate from one task to another accompanied by a stack region appropriate to the thread.



(a) UNIX (b) Mach (c) ToM

Figure 2: Programming Models of Current OSs

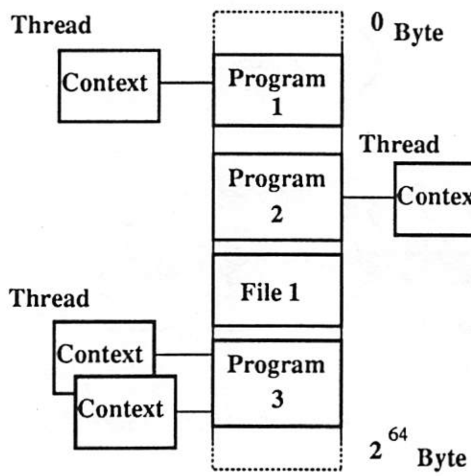


Figure 3: Programming Model of New Micro Kernel

3 Memory Model

3.1 Single Virtual Storage (SVS)

In UNIX, each process has an address space independently and a user can indicate where to allocate the program in a 32-bit address space. On the contrary, this micro

kernel holds a huge single address space and takes on the following properties.

- A shared memory or library is allocated at the same address, viewing from every thread, because procedures or files are mapped in a single address space.
- The linker allocates a program at an adequate fixed address among free memory sections when it is compiled. A dynamic link can be achieved simply. A file is also allocated as a fixed address, when it is created.

This micro kernel can take greater advantage of a cache system, since it does not remap the virtual address when switching threads for the following reasons.

- A virtual cache system can achieve faster access compared with a physical cache system. Single virtual addressing can take advantage of a virtual cache system without causing an aliasing problem because it provides a unique virtual address to all the system resources.
- Context (Space) switching is said to be an expensive process in UNIX, and a large proportion of time is wasted for maintaining cache consistency (Mogul et al. [5]) which is avoidable in the SVS.

3.2 One Level Storage (OLS)

One level storage is where the kernel treats the file space not as a special space but as a normal address space. This is realized partially in 4.3BSD UNIX (Leffler et al. [6]) by *mmap* system call. This merit is to access the file data first through not using system calls, such as read or write, which that is costly. But this needs a large address space for the mapping file space to the address space. A conventional OS doesn't support sufficient space, and a portion of the file data can be mapped on the space. A huge address space can be used on this kernel and programs can map all files on the same space

There are other merits for adopting OLS.

- Dynamic linking can be easy supported for all library files (object files) mapped on SVS.
- A micro kernel structure is simple because there is no difference between processes and files at the micro kernel level.

Dynamic linking is effective *rapid prototyping* with a reuse of programs coded in advance. It is supported by several modern systems, such as SunOS 4.x and OS/2, as well as Multics (Organick[7]). However, this function in the modern system is limited to specially defined libraries. The limitation is mainly caused by narrow address spaces. A huge SVS can obviate such a limitation.

The problem regarding access from unauthorized threads or programs is solved with the newly designed MMU described in the following.

4 High Speed RPC on SVS

A client-server programming style is applicable to a distributed environment on workstations, but limits the performance on ordinary OSs. The reason is that the ordinary way for executing a job involves the following steps.

- Allocate each client and server to a respective process.
- A client requests a service to the server using an IPC, such as pipe, socket, message passing, RPC, and shared memory.
- This instigates OS services and address space switching between the two processes.

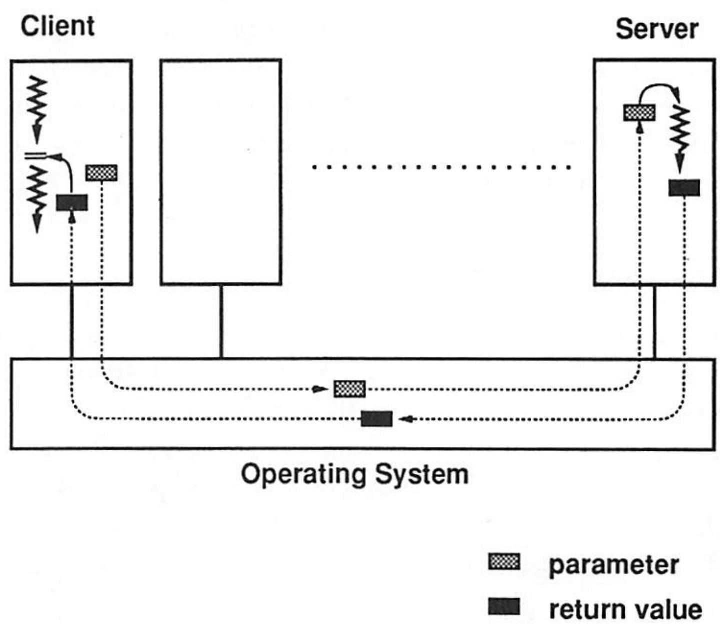


Figure 4: Ordinary RPC

Usually, OS services waste time, and address switching damages the logical cache performance.

On the other hand, this micro kernel executes a client-server program as follows.

- Load a client and a server procedure in the SVS
- Allocate a thread to the client
- Execute related procedures without address switching

Because there are both client and server procedure on an SVS, the micro kernel doesn't need to switch the address space and flash out the logical cache data.

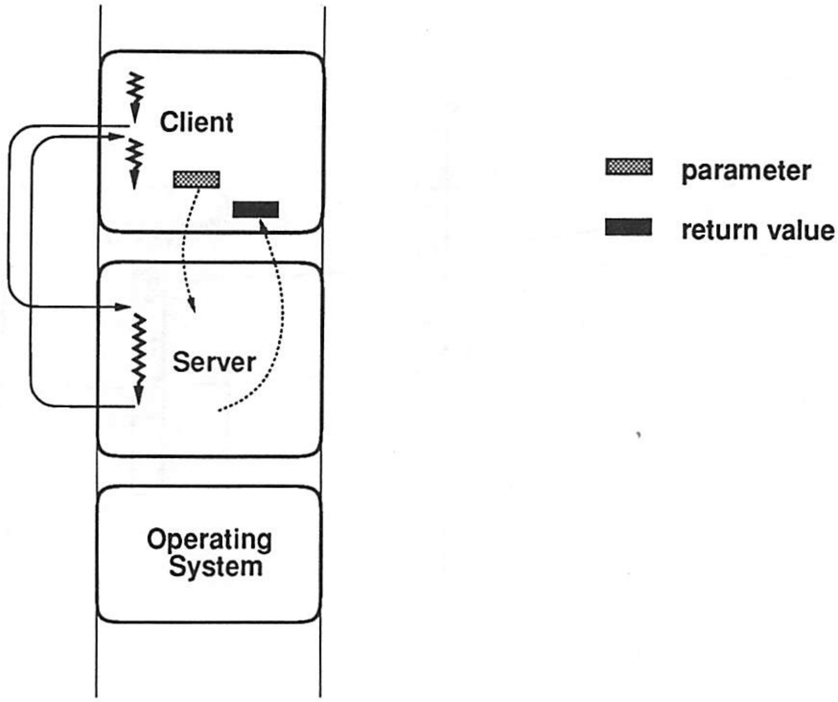


Figure 5: RPC on SVS

5 Network Transparency on OLS

It is important that the OS offer network transparency on a distributed environment, especially for a workstation environment. This micro kernel supports the network transparent SVS by the OLS. A pager maps an address space with a backing storage. This micro kernel supports an external pager, the same as Mach or ToM. The micro kernel detects any page fault access and calls the external pager only with the page fault message. It doesn't handle the network transparency processing. It is handled by the pager routines in the file servers. For a huge SVS, all files in a *cluster* (host group connected by a high speed LAN) are shared in the SVS.

A consistency problem, one of the problems in a distributed file system, still exists and isn't solved by this micro kernel.

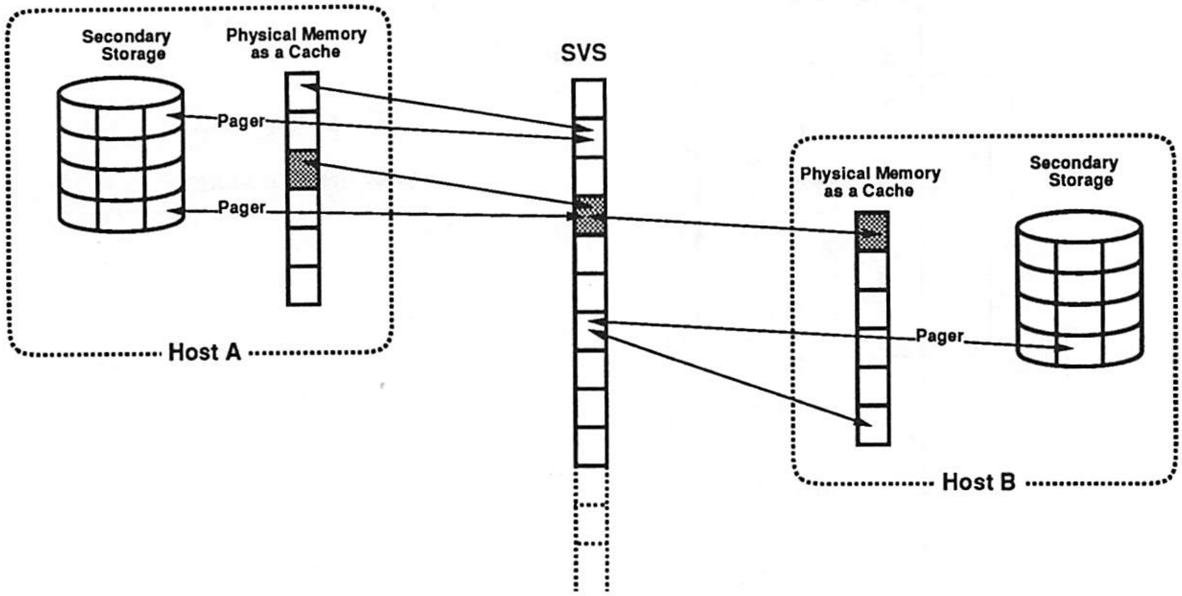


Figure 6: Network Shared Memory

6 Memory Protection Mechanism

6.1 Two Kinds of ACLs on the Memory Section

Owing to the OLS, all programs and files exist in the same address space, and address switching will never occur. This memory model realizes user-level RPC. On the other hand, it cannot detect and doesn't prevent illegal access without kernel support. This kernel provides two kinds of memory protection mechanisms as operations for memory section attributes. Each memory section possesses two type of access control lists (ACLs) whose entries respectively consist of (memory section identifier, protection attribute) and (thread identifier, protection attribute). These respectively mean:

- *From which memory section* the specified memory section is to be accessed.
- *By which thread* the specified memory section is to be accessed.

Protection attributes are such as read, write, and execute. With a combination of these ACLs, many kinds of memory section, such as the T(ext) section, D(ata) section, S(tack) section, F(ile) section and G(ate) section, can be created flexibly. A D Section is accessed only through a specified T Section. An S Section is accessed only by a specified thread. An F Section is accessed by a file server or an authorized thread.

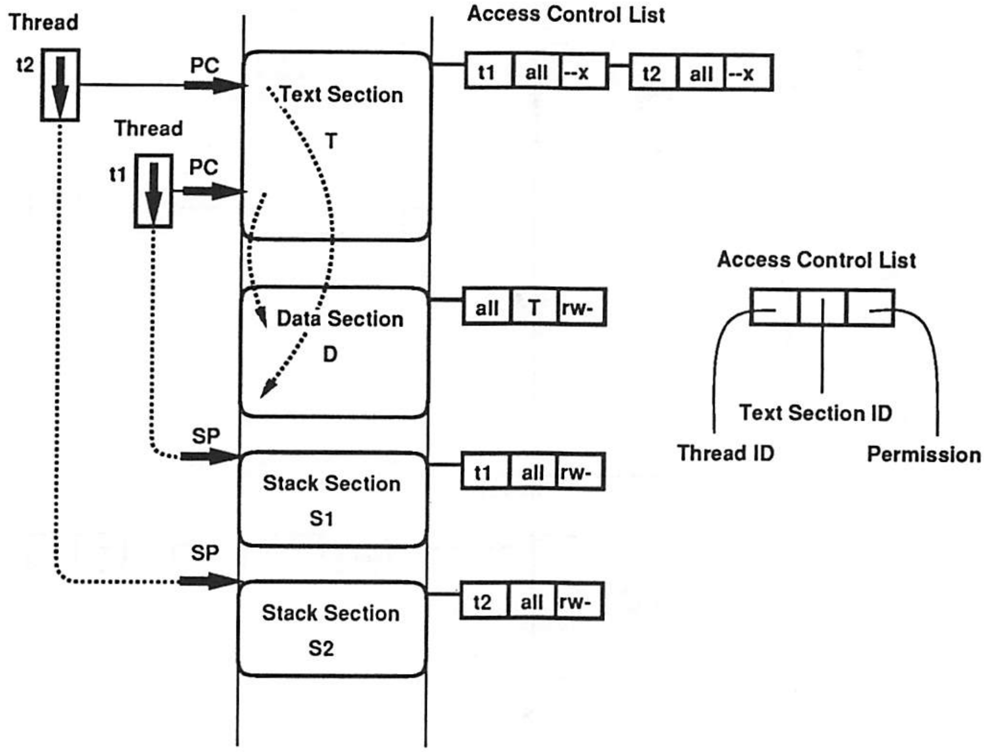


Figure 7: Memory Section Category

6.2 Gate Section

The G Section is conceptually the same as a Multics gate . A G Section is introduced to implement user-level RPC without kernel invoking, and it involves a branch list of service entries to jump. When a thread jumps from a client procedure to a server procedure, it cannot jump directly to the destination code address by way of the server procedure G Section. It first jumps from a client procedure to the server G Section and then jumps from the G Section to the specified service entry. This indirect jump mechanism prevents illegal address access to server procedure and allows only authorized threads to jump to a specified entry, since only authorized threads are given the capability to access the G Section.

7 Hardware Requirement

7.1 64-bit Address Space

This micro kernel requires a processor that supports at least a 64-bit addressing mode to use a huge address space. Recently, some 64-bit processors have been announced, but they don't support full range address space. For example, the current MIPS R4000 supports only a 42-bit address space. How this micro kernel design can work well can be tested on the narrow 42-bit address space processor. A 64-bit address

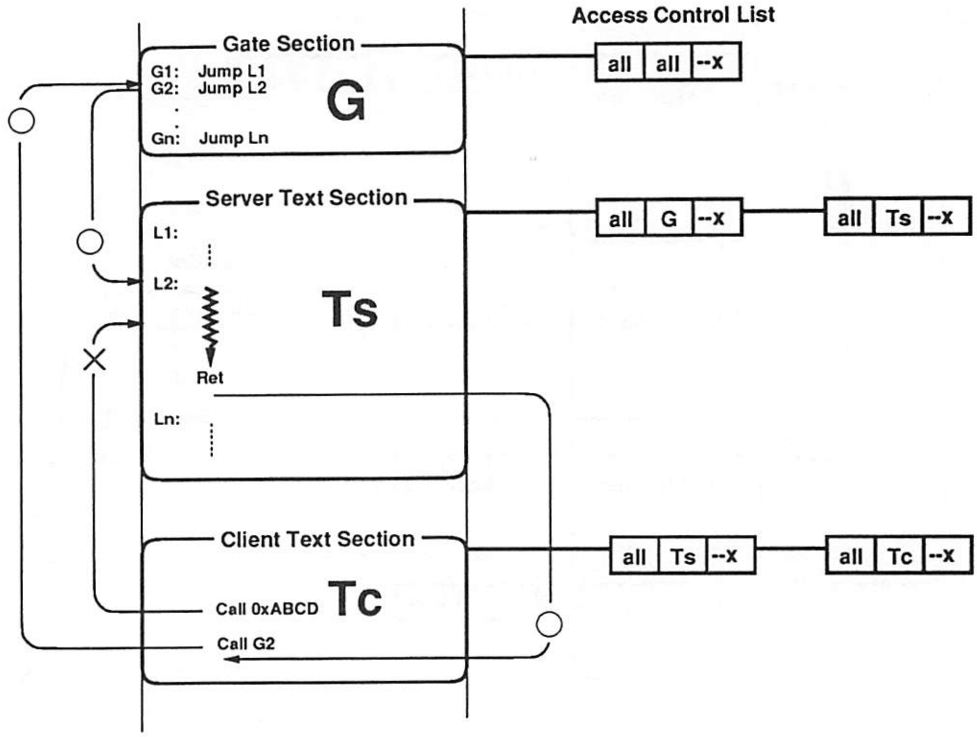


Figure 8: Gate Section

space processor will appear soon.

7.2 Rich-Functioned MMU

This micro kernel assumes an additional hardware support to achieve a fine-grain page-based memory protection mechanism to support a safety user-level RPC, though conventional MMUs cannot support both functions:

- Loading multiple procedures in a single space
- Detecting an illegal cross-procedure memory access

The newly designed MMU handles two kinds of ACLs (access control by the memory section and by the thread) besides information on an ordinary page table entry (PTE).

Figure 9 shows the PTE structure. An idea to speed up checking the ACL is to cache several ACL entries in the translation look aside buffer(TLB). The checking hardware has the same number of comparators as the cache, and compares all cached entries with the accessing thread and memory section simultaneously. An ACL fault invokes the pager for searching the rest of the ACLs stored in the memory and causes a swap for a cached entry.

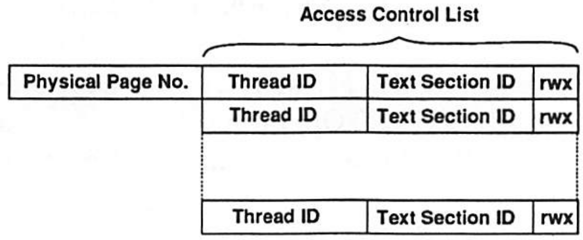


Figure 9: Rich-Functioned MMU

8 Conclusion

This micro kernel aims at a huge 64-bit address space, which is a trend for next generation micro processors, with

- *SVS* loading all procedures in one virtual space.
- *OLS* loading files in the virtual space.

These distinctive memory model aspects bring the following merits to both users and system builders.

- *Performance advantage* in shared service and shared bulk data processing because a client procedure calls a server procedure and accesses a file without invoking kernel calls.
- *Portability advantage* because of the simplified micro kernel architecture with only two abstractions, a thread and a memory section.

This micro kernel is still in the system design stage and applied to a distributed environment. The first step is to implement the Mach system call interface on the micro kernel to install a UNIX environment.

The authors will confirm this system design on an Intel 386 or 486 processor. Its segment function is used as a substitute for a rich-functioned MMU.

The authors also plan to develop a server that takes the maximal advantage from the micro kernel features mentioned previously.

References

[1] John R. Mashey, "64-bit Computing", *Byte*, pp.135-142, September 1991.

[2] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M.: "Mach: A New Kernel Foundation For UNIX Development", *Proceedings of USENIX 1986 Summer Conference*, pp.93-112, 1986.

[3] Brian N. Bershad, Thomas Anderson, Edward Lazowska and Henry Levy, "User-Level Interprocess Communication for Shared Memory Multiprocessors", *ACM Transaction on Computer System*, Vol.9, No.2, pp.175-198, May 1991.

[4] Arai, J., Tuiki, H., Hagino, T., Hattori, T. and Morishima, A.: "A Distributed Operating System ToM" (in Japanese), Kyoto University, Research Institute for Mathematical Sciences, Technical Report RIMS–675, 1989.

[5] Jeffrey C. Mogul and Anita Borg, "The Effect of Context Switches on Cache Performance", *ASPLOS-IV Proceedings*, pp.75-85, April 1991.

[6] Leffler, S., McKusick, M., Karels, M. and Quarterman, J.: "The Design and Implementation of the 4.3BSD UNIX Operating System ", Addison-Wesley, 1989.

[7] Elliot I. Organick, "The Multics System: An Examination of Its Structure", MIT, 1972.

The KeyKOS® Nanokernel Architecture

Alan C. Bomberger
A. Peri Frantz
William S. Frantz
Ann C. Hardy
Norman Hardy
Charles R. Landau
Jonathan S. Shapiro

ABSTRACT

The KeyKOS nanokernel is a capability-based object-oriented operating system that has been in production use since 1983. Its original implementation was motivated by the need to provide security, reliability, and 24-hour availability for applications on the Tymnet® hosts. Requirements included the ability to run multiple instantiations of several operating systems on a single hardware system. KeyKOS was implemented on the System/370, and has since been ported to the 680x0 and 88x00 processor families. Implementations of EDX, RPS, VM, MVS, and UNIX have been constructed. The nanokernel is approximately 20,000 lines of C code, including capability, checkpoint, and virtual memory support. The nanokernel itself can run in less than 100 Kilobytes of memory.

KeyKOS is characterized by a small set of powerful and highly optimized primitives that allow it to achieve performance competitive with the macrokernel operating systems that it replaces. Objects are exclusively invoked through protected capabilities, supporting high levels of security and intervals between failures in excess of one year. Messages between agents may contain both capabilities and data. Checkpoints at tunable intervals provide system-wide backup, fail-over support, and system restart times typically less than 30 seconds. In addition, a journaling mechanism provides support for high-performance transaction processing. On restart, all processes are restored to their exact state at the time of checkpoint, including registers and virtual memory.

This paper describes the KeyKOS architecture, and the binary compatible UNIX implementation that it supports.

1. Introduction

This paper describes the KeyKOS nanokernel, a small capability-based system originally designed to provide security sufficient to support mutually antagonistic users. KeyKOS consists of the nanokernel, which can run in as little as 100 Kilobytes of memory and includes all of the system privileged code, plus additional facilities necessary to support operating systems and applications. KeyKOS presents each application with its own abstract machine interface. KeyKOS applications can use this abstract machine layer to implement KeyKOS services directly or to implement other operating system interfaces. Implementations of EDX, RPS, VM/370, an MVS subset, and UNIX have been ported to the KeyKOS platform using this facility.

Tymshare, Inc. developed the earliest versions of KeyKOS to solve the security, data sharing, pricing, reliability, and extensibility requirements of a commercial computer service in a network environment.

KeyKOS is a registered mark of Key Logic, Inc.

Tymnet is a registered mark of British Telecom, Inc.

UNIX is a registered mark of AT&T Bell Laboratories, Inc.

Development on the KeyKOS system began in 1975, and was motivated by three key requirements: accounting accuracy that exceeded any then available; 24-hour uninterrupted service; and the ability to support simultaneous, mutually suspicious time sharing customers with an unprecedented level of security. Today, KeyKOS is the only commercially available operating system that meets these requirements.

KeyKOS began supporting production applications on an IBM 4341 in January 1983. KeyKOS has run on Amdahl 470V/8, IBM 3090/200 (in uniprocessor System/370 mode), IBM 158, and NAS 8023. In 1985, Key Logic was formed to take over development of KeyKOS. In 1988, Key Logic began a rewrite of the nanokernel in C. After 10 staff months of effort a nanokernel ran on the ARIX Corporation 68020 system, and the project was set aside. The project resumed in July of 1990 on a different processor, and by October of 1990 a complete nanokernel was running on the Omron Luna/88K. The current nanokernel contains approximately 20,000 lines of C code and less than 2,000 lines of assembler code.

This paper presents the architecture and design of the KeyKOS nanokernel, and the UNIX system that runs on top of it. In the interest of a clear presentation of the KeyKOS architecture, we have omitted a description of the underlying kernel implementation.

2. Architectural Foundations

KeyKOS is founded on three architectural concepts that are unfamiliar to most of the UNIX community: a stateless kernel, single-level store, and capabilities. Our experience indicates that understanding a single-level store model requires a fundamental shift in perspective for developers accustomed to less reliable architectures. It therefore seems appropriate to present these concepts first as a foundation on which to build the balance of the KeyKOS architectural description.

Stateless Kernel

An early decision in the KeyKOS design was to hold no critical state in the kernel. All nanokernel state is derived from information that persists across system restarts and power failures. For reasons of efficiency, the nanokernel does reformat state information in private storage. All private storage is merely a cache of the persistent state, and can be recycled at any time. When the discarded information is needed again, it is reconstructed from the information in nodes and pages (which are described below)

As a consequence, the nanokernel performs no dynamic allocation of kernel storage. This has several ramifications:

- The kernel is faster, since no complicated storage allocation code is ever run.
- The kernel never runs out of space.
- There is no nanokernel storage (such as message queues) that must be a part of the checkpoint.

The absence of dynamic allocation means that there can be no interaction between dynamic allocation strategies, which is the predominant source of deadlock and consistency problems in most operating systems.

The system outside the nanokernel is completely described by the contents of nodes and pages (see below), which are persistent. This state includes files, programs, program variables, instruction counters, I/O status, and any other information needed to restart the system.

In addition, the ability to recover all run-time kernel data from checkpointed state means that an interruption of power does not disrupt running programs. Typically, the system loses only the last few seconds of keyboard input. At UNIFORUM '90, Key Logic pulled the plug on our UNIX system on demand. Within 30 seconds of power restoration, the system had resumed processing, complete with all windows and state that had previously been on the display. We are aware of no other UNIX implementation with this feature today.

Single-Level Store

KeyKOS presents a persistent single-level store model. To the KeyKOS application, all data lives in persistent virtual memory. Only the nanokernel is aware of the distinction between main memory and disk pages. Periodic system-wide checkpoints guarantee the persistence of all system data. The paging system is tied to the checkpoint mechanism, and is discussed in the section on checkpointing, below. Persistence extends across system shutdown and power failure. Several IBM 4341 systems ran for more than three years across power failures without a logical interruption of service.

Like memory pages, KeyKOS applications are persistent. An application continues to execute until it is explicitly demolished. To the application, the shutdown period is visible only as an unexplained jump in the value of the real time clock, if at all. As a result, the usual issues surrounding orderly startup and shutdown do not apply to KeyKOS applications. Most operating systems implement a transient model of programs; persistence is the exception rather than the rule. A client operating system emulator may provide transient applications by dismantling its processes when they terminate.

The single-level store model allows far-reaching simplifications in the design of the KeyKOS system. Among the questions that the nanokernel does *not* have to answer are:

- How does the system proceed when it runs out of swap space? (It checkpoints.)
- How does the kernel handle the tear-down of a process? (It doesn't.)
- How is kernel state retained across restarts? (The kernel contains no state that requires checkpointing.)

Each of these areas is a source of significant complexity in other systems, and a consequent source of reliability problems.

Capabilities

KeyKOS is a capability system. For brevity, KeyKOS refers to capabilities as ***keys***. Every object in the system is exclusively referred to by one or more associated keys. Keys are analogous in some ways to Mach's ports. KeyKOS entities call upon the services of other entities by sending messages via a key. Message calls include a kernel-constructed ***return key*** that may be used by the recipient to issue a reply. Messages are most commonly exchanged in an RPC-like fashion.

What sets KeyKOS apart from other microkernels is the total reliance on capabilities without any other mechanisms. There are no other mechanisms that add complexity to the ideas or to the implementation. Holding a key implies the authority to send messages to the entity or to pass the key to a third party. If A does not have a key for B, then A cannot communicate with B. Applications may duplicate keys that they hold, but the creation of keys is a privileged operation. The actual bits that identify the object named by a key are accesible only to the nanokernel.

Through its use of capabilities and message passing, KeyKOS programs achieve the same encapsulation advantages of object-oriented designs. Encapsulation is enforced by the operating system, and is

available in any programming language. It is the complete security of this information hiding mechanism that makes it possible to support mutually suspicious users.

A fundamental concept in KeyKOS is that a program should obey the "principle of least privilege". To that end, the design of KeyKOS gives objects *no* intrinsic authority, and relies totally upon their keys to convey what authority they have. Using these facilities, the system is conveniently divided into small modules, each structured so as to hold the minimal privilege sufficient for its operation.

Entities may be referred to by multiple, distinct keys. This allows an entity that communicates with multiple clients to grant different access rights to the clients. Every key has an associated 8-bit field that can be used by the recipient to distinguish between clients. When the entity hands out a key, it can set the field to a known value. Because all messages received by the entity include the 8-bit value held in the key, this mechanism can be used to partition clients into service classes or privilege levels by giving each class a different key.

It is worthwhile to contrast this approach with the ring-structured security model pioneered in Multics and propagated in the modern Intel 80x86 family. The capability model is intrinsically more secure. A ring-structured security policy is not powerful enough to allow a subsystem to depend on the services of a subsystem with lesser access rights. Ring policies intrinsically violate the principle of least privilege. In addition, ring-based security mechanisms convey categorical authority: any code running in a given layer has access to all of the data in that layer. Capability systems allow authority to be minimized to just that required to do the job at hand.

Using a capability model offers significant simplifications in the nanokernel. Among the questions that the nanokernel does *not* have to answer are:

- Does this user have the authority to perform this operation? (Yes – if you hold the key you can send the message.)
- How do I allocate enough kernel memory to perform name resolution on a variable length name? (The kernel never deals with names, only keys.)
- Where does this file name get inserted in this directory? (The nanokernel does not deal with file names or directories.)

Because the nanokernel has no naming mechanism other than capabilities, entity naming is intrinsically decentralized. As a result, extending KeyKOS to multiprocessors is straightforward. KeyKOS applications cannot tell if they are running on a uniprocessor or a multiprocessor.

3. Major Nanokernel Features

The nanokernel includes all of the supervisor-mode code of the system. The entire kernel is implemented in approximately 20,000 lines of reasonably portable C code, and 2,000 lines of 88x00 assembly code. Of the assembly lines, 1,000 lines are in the context switch implementation. This compiles to roughly 60 Kilobytes of executable code. While running, the nanokernel requires as little as 100 Kilobytes of main memory.

The nanokernel is the only portion of the system that interprets keys. No other program has direct access to the bits contained in the keys, which prevents key forgery. In addition, the nanokernel includes code that defines the primitive system objects. These objects are sufficient to build the higher-level abstractions supported by more conventional operating systems. The nanokernel provides:

- multiprogramming support, primitive scheduling, and hooks for more sophisticated schedulers running as applications;
- a single-level store, as discussed above;
- separate virtual address space(s) for each KeyKOS process;
- redundant disk storage for system-critical information;
- a system-wide checkpoint-restart feature;
- journaling pages exempt from checkpoint for database and transaction processing support;
- keys by which messages are sent from one application to another;
- primitive and limited access to individual I/O devices;
- interpretation of keys that hides the location of the object on disk or in main memory.

During normal operation, KeyKOS executes a system-wide checkpoint every few minutes to protect from power failures, most kernel bugs, and detected hardware errors. Both data *and processes* are checkpointed. All run-time state in the nanokernel can be reconstructed from the checkpoint information. Except for the initial installation, the system restarts from the most recent checkpoint on power up.

In addition to local checkpoint support, the nanokernel provides for checkpoints to magnetic tape or remote hot-standby systems. This allows a standby system to immediately pick up execution in the event of primary system failure.

4. Fundamental KeyKOS Objects

The KeyKOS kernel supports six types of fundamental objects: ***devices***, ***pages***, ***nodes***, ***segments***, ***domains***, and ***meters***.

Devices

The nanokernel implements low-level hardware drivers in privileged code. The supervisor-mode driver performs message encapsulation and hardware register manipulation. Except where performance compels otherwise, KeyKOS applications implement the actual device drivers.

Pages

The simplest KeyKOS object is the page. Page size is dependent on the underlying hardware and storage architectures, but in all current implementations is 4 Kilobytes. Every page has one or more persistent locations on some disk device, known as its ***home location***. The KeyKOS system manages a fixed number of pages that are allocated when the system is first initialized. This number can be increased by attaching additional mass storage devices to the system.

A page is designated by one or more ***page keys***. Pages honor two basic message types: read, and write. When pages are mapped into a process address space, loads and stores to locations in a page are isomorphic to read and write messages on the page key. When a message is sent to a page that is not in memory, the page is transparently faulted in from backing store so that the operation can be performed.

Applications that perform dynamic space allocation hold a key to a ***space bank***. Space banks are used to manage disk resource allocation. The system has a master space bank that holds keys to all of the pages and nodes in the system.[1] One of the operations supported by space banks is creating subbanks, which are subbanks of the master space bank. If your department has bought the right to a megabyte of storage, it is given a key to a space bank that holds 256 page keys. Space banks are a type of domain.

Nodes

A node is a collection of keys. All keys in the system reside in nodes. A ***node key*** conveys access rights to a node, and can be used to insert or remove keys from a node. Like pages, nodes can be obtained from space banks. In all current KeyKOS implementations, a node holds precisely 16 keys.

Nodes are critical to the integrity of the system. The KeyKOS system vitally depends on the data integrity of node contents. As a result, all nodes are replicated in two (or more) locations on backing store. In keeping with the general policy of not performing dynamic allocation in the kernel, and because the integrity requirements for nodes are so critical, KeyKOS does not interconvert nodes and pages.

Segments

A segment is a collection of pages or other segments. Segments are used as address spaces, but also subsume the function of files in a conventional operating system. Segments can be combined to form larger segments. Segments may be sparse; they do not necessarily describe a contiguous range of addresses.

Nodes are the glue that holds segments together. KeyKOS implements segments as a tree of nodes with pages as the leaves of the tree. This facilitates efficient construction of host architecture page tables. Because nodes and pages persist, so do segments. The system does not need to checkpoint page table data structures because they are built exclusively from the information contained in segments.

Meters

Meters control the allocation of CPU resources. A ***meter key*** provides the holder with the right to execute for the unit of time held by the meter. The KeyKOS kernel maintains a ***prime meter*** that represents the time interval from the present until the end of time. Like space banks, meters can be subdivided into submeters. Every running process holds a meter key that authorizes the process to execute for some amount of time.

KeyKOS processes can be preempted. Holding a key to a meter that provides 3 seconds of CPU time does not guarantee that the process will run for 3 contiguous seconds. In the actual KeyKOS implementation, time slicing is enforced by allowing a process to run for the minimum of its entitled time or the time slice unit. Political scheduling policies may be implemented external to the kernel.

Domains

Domains perform program execution services. They are analogous to the virtual processors of the POSIX threads mechanism. It was a design goal not to restrict the architecture available to the user. A consequence is that KeyKOS supports virtual machines. Domains model all of the non-privileged state of the underlying architecture, including the general purpose register set, floating point register set, status

---

1 The system can support multiple master space banks. In a B3 implementation, system pages would be partitioned into multiple security classes, and there would be one master space bank for each class.

registers, instruction set architecture, etc. A domain interprets a program according to the hardware user-mode architecture. Domains are machine-specific, though we have considered the implementation of domains that perform architecture emulation (e.g. for DOS emulation on a RISC machine).

In addition to modeling the machine architecture, domains contain 16 general key slots and several special slots. The 16 general slots hold the keys associated with the running program. When a key occupies one of the slots of a domain, we say that the program executing in that domain holds the key. One of the special slots of the domain is the ***address slot***. The address slot holds a segment key for the segment that is acting as the address space for the program. On architectures with separate instruction and data spaces, the domain will have an address slot for each space. Each domain also holds a meter key. The meter key allows the domain to execute for the amount of time specified by the meter.

KeyKOS processes are created by building a segment that will become the program address space, obtaining a fresh domain, and inserting the segment key in the domain's address slot. The domain is created in the *waiting* state, which means that it is waiting for a message. A threads paradigm can be supported by having two or more domains share a common address space segment.

Because domain initialization is such a common operation, KeyKOS provides a mechanism to generate "prepackaged" domains. A ***factory*** is an entity that constructs other domains. Every factory creates a particular type of domain. For example, the queue factory creates domains that provide queuing services. An important aspect of factories is the ability of the client to determine their trustworthiness. It is possible for a client to determine whether an object created by a factory is secure. Understanding factories is crucial to a real understanding of KeyKOS, but in the interest of brevity we have elected to treat factories as "black boxes" for the purposes of this paper. To understand the UNIX implementation it is sufficient to think of factories as a mechanism for cheaply creating domains of a given type.

5. Message Passing

The most important operation supported by the nanokernel is message passing. Messages sent from one domain to another involve a context switch. In order to encourage the separation of applications into components of minimal privilege, the nanokernel's message transfer path has been carefully optimized. The KeyKOS inter-domain message transfer path ranges from 90 instructions on the System/370 to 500 cycles on the MC88x00.

Messages are composed of a parameter word (commonly interpreted as a method code), a string of up to 4096 bytes, and four keys. A domain constructs a message by specifying an integer, contiguous data from its address segment, and the keys to be sent. Only keys held by the sender can be incorporated into a message. Once constructed, the message is sent to the object named by a specified key. Sending a message is sometimes referred to as key invocation.

KeyKOS supplies three mechanisms for sending messages. The ***call*** operation creates a ***resume key***, sends the message to the recipient, and waits for the recipient to reply using the message's resume key. While waiting, the calling domain will not accept other messages. A variant is ***fork***, which sends a message without waiting for a response. The resume key is most commonly invoked using a ***return*** operation, but creative use of call operations on a resume keys can achieve synchronous coroutine behavior. The return operation sends a message and leaves the sending domain available to respond to new messages. All message sends have copy semantics.

The nanokernel does not buffer messages; a message is both sent and consumed in the same instant. If necessary, invocation of a key is deferred until the recipient is ready to accept the message. Message buffering can be implemented transparently by an intervening domain if needed. The decision not to buffer messages within the nanokernel was prompted by the desire to avoid dynamic memory allocation, limit I/O overhead, keep the context switch path length short, and simplify the checkpoint operation.

A message recipient has the option to selectively ignore parts of a message. It may choose to accept the parameter word and all or part of the byte string without accepting the keys, or accept the parameter word and the keys without the data.

6. Checkpointing and Journaling

KeyKOS provides for regular system-wide checkpoints and individual page journaling. Checkpoints guarantee rapid system restart and fail-over support, while journaling provides for databases that must make commit guarantees.

The Checkpoint Mechanism

The KeyKOS nanokernel takes system-wide checkpoints every few minutes. Checkpoint frequency can be adjusted by the administrator at any time without interruption of service.

The KeyKOS system maintains two disk regions as checkpoint areas. When a checkpoint is taken, all processes are briefly suspended while a rapid sweep is done through system memory to locate modified pages. No disk I/O is done while processes are frozen. Once the sweep has been done, processes are resumed and all modified pages are written to the current checkpoint area. Once the checkpoint has completed, the system makes the other checkpoint area current, and begins migrating pages from the first checkpoint area back to their home locations. Checkpoint frequency is automatically tuned to guarantee that the page migration process will complete before a second checkpoint is taken. Because the migration process is incremental, a power failure during migration never leads to a corrupt system.

An implementation consequence of this approach to checkpointing is unusually efficient disk bandwidth utilization. Checkpoint, paging, and page migration I/O is optimized to take advantage of disk interleave and compensates for arm latencies to minimize seek delays. This accounts for all page writes. The aggregate result is that KeyKOS achieves much higher disk efficiency than most operating systems. If the system bus is fast enough, KeyKOS achieves disk bandwidth utilization in excess of 90% on all channels.

It is worth emphasizing that the checkpoint is not simply of files, but consists of all processes as well. If an update of a file involves two different pages and only one of the pages has been modified at the time of the checkpoint, the file will not be damaged if the system is restarted. When the system is restarted the process that was performing the update is also restarted and the second page of the file is modified as if there had been no interruption. A power outage or hardware fault does not leave the system in some confused and damaged state. The state at the last checkpoint is completely consistent and the system may be restarted from that state without concern about damaged files.

The Journaling Mechanism

For most applications, it is acceptable for the system as a whole to lose the last few minutes work after a power outage. Transaction processing and database systems require the additional ability to commit individual pages to permanent backing store on demand. Using the journaling mechanism, a domain may request that changes to a particular page be synchronously committed to permanent storage. If a system failure occurs between the commit and the next completed checkpoint, the journaled page will remain committed after the system restarts. It is the responsibility of the requesting domain to see to the semantic consistency of such pages.

The journaling mechanism commits pages by appending them to the most recent committed checkpoint. As a result, journaling does not lead to excessive disk arm motion. A curious consequence of this implementation is that transaction performance under KeyKOS *improves* under load.[2] This is due to locality at two levels. As load increases, it becomes common for multiple transactions to be committed by a single page write. In addition, performing these writes to the checkpoint area frequently allows the journaling facility to batch disk I/O, minimizing seek activity. The KeyKOS transaction system significantly exceeds the performance of competing transaction facilities running on the same hardware. CICS, for example, is unable to commit multiple transactions in a single write.

7. Exception Handling

Process exceptions are encapsulated by the nanokernel and routed to a user-level handler known as a ***keeper***. The keeper technology of KeyKOS brings all exception policy to application level programs outside of the nanokernel. A keeper is simply a domain that understands the exception messages delivered by the kernel; it is in all regards an ordinary domain. Since the UNIX implementation relies heavily on the Domain Keeper technology, the ideas and specifications concerning Keepers will be discussed before we delve into the UNIX specifics.

Recall that a KeyKOS application has an address space, a domain, and a meter. Each of these objects holds a start key to an associated domain known as its ***keeper***. When the process performs an illegal, unimplemented, or privileged instruction, the error is encapsulated in a message which is sent to the appropriate keeper, along with the keys necessary to transparently recover or abort the application. The keeper may terminate the offending program, supply a correct answer and allow execution to continue, or restart the offending instruction.

Each segment has an associated ***segment keeper***. The segment keeper is a KeyKOS process that is invoked by the kernel when an invalid operation, such as an invalid reference or protection violation, is performed on a segment. Page faults are fielded exclusively by the nanokernel.

By appropriate use of a ***meter keeper***, more sophisticated scheduling policies can be implemented. The meter keeper is invoked whenever the meter associated with a domain times out. A thread supervisor might implement a priority scheduling policy by attaching the same meter keeper to all threads, and having the meter keeper parcel out time to the individual threads according to whatever policy seemed most sensible.

The most interesting keeper for this paper is the ***domain keeper***. The domain keeper is invoked when a trap or exception is taken. When a domain encounters an exception (system call, arithmetic fault, invalid operation, etc.) the domain stops executing and the domain keeper receives a message. The message contains the non-privileged state of the domain (its registers, instruction counter, etc.), a domain key to the domain, and a form of resume key that the keeper can use to restart the domain. When the faulting domain is restarted, it resumes at the instruction pointed to by the program counter. If necessary, the domain keeper can adjust the PC value of the faulting domain before resumption.

8. A KeyKOS-Based UNIX Implementation

In July of 1990, Key Logic undertook to produce a binary-compatible prototype UNIX implementation for the Omron Luna/88K. The effort had two principle goals. The first was to rapidly construct a system that could run existing Omron application binaries. Based on Mach 2.5, the Omron implementation provides a reasonably complete version of the Berkeley UNIX system, including the X11r4 windowing

---

2 Up to a point. There ain't no such thing as a free lunch.

system. KeyNIX was implemented by a single developer over a six month period, without reference to the UNIX source code. The implementation was partly based on an earlier Minix port that had been built for KeyKOS on the System/370.

Our experience in implementing other systems was that breaking an application into separate function-oriented domains simplified the application enough to improve overall performance. A second goal of the KeyNIX implementation was to learn where such decomposition into separate domains would cause performance degradation. In several areas, multi-domain implementations were tried where the problem area was clearly a boundary case in order to explore the limitations of the domain paradigm.

UNIX Services

Broadly speaking, the UNIX system provides the following services:

- Process management (fork, exec, exit, kill),
- File system and namespace services (open, link),
- I/O services (read, write, stat, ...)
- Timing facilities (sleep, nap, sometimes socket)
- Messaging (sockets, pipes)
- Memory management (mmap, mprotect)
- Signals
- Device support
- Networking (TCP/IP, NFS)

With the exception of networking, KeyNIX implements all of these services. Adding networking support would be straightforward, but was not part of the prototype effort.

9. Structure of KeyNIX

Under KeyNIX, every UNIX process runs as a KeyKOS domain with a segment as its address space. A standard KeyKOS segment keeper is used to manage stack and heap growth within the address space segment. From the outside, the UNIX process model is essentially unchanged. No KeyNIX code is mapped in with the application, nor is special linking required. The application address spaces are bit for bit identical. This severely penalizes all trivial system calls, and is a significant departure from the implementations used by other microkernels. The penalty could be eliminated in a dynamic-library based standard such as System V Release 4.

To support UNIX processes, we implemented a domain keeper, known as the ***UNIX Keeper***. The UNIX keeper interprets the system call and either manages the call itself or directs request to other domains for servicing. The implementation includes a number of cooperating domains, is shown in Figure 1. The gray box surrounds the domains and segments that are replicated for each UNIX process.

Each of these domains in turn depends on other domains provided by the KeyKOS system. For example, a small integer allocator domain is used to allocate monotonically increasing inode numbers. To simplify

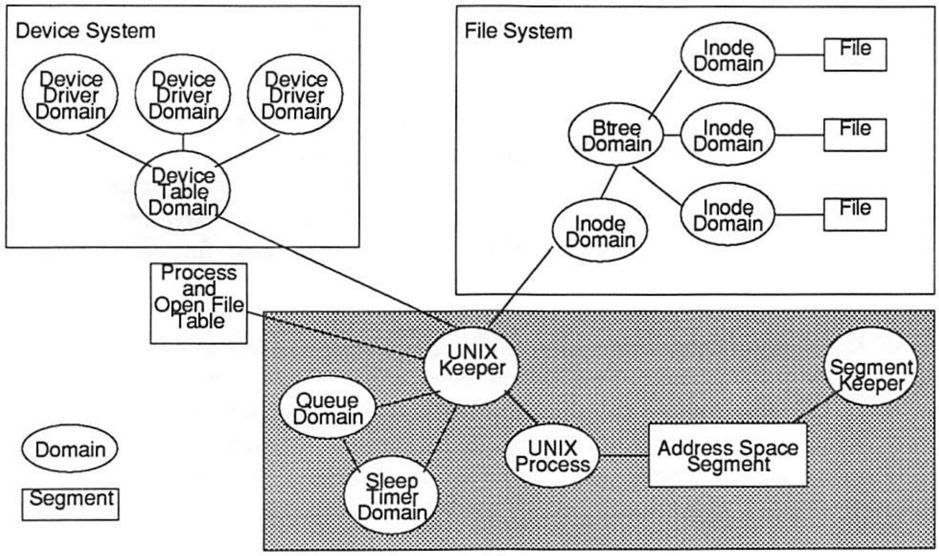


Figure 1: Structure of the UNIX Implementation

the picture, domains that are not essential to understanding the structure of the UNIX implementation have been omitted.

One Kernel per Process

An unusual aspect of the KeyNIX design is that every UNIX process has a dedicated copy of the UNIX Keeper. When a process forks, the UNIX Keeper is replicated along with the process. By providing a separate UNIX keeper to each UNIX application, the scope of UNIX system failures is reduced to a single process. If a given UNIX process *does* manage to crash its copy of the operating system, no other processes are impacted. An individual kernel is very hard to crash. To crash the entire UNIX system essentially requires physical abuse of the machine or its power supply.

State that must be shared between multiple UNIX keepers, including the process table and open file table, is kept in a segment shared by all UNIX Keepers. Each process has a description block (a process table entry) that describes the process' address space, open files, and signal handling. Process table entries contain chains of child processes and pointers to the parent process table entry. Each open file has an entry in the Open File Table which keeps track of the number of processes that have the file open, the attributes of the file, and a pointer to the data structures that buffer the file data in memory.

The UNIX keeper implements UNIX process and memory management services by calling directly on the underlying KeyKOS services. The nanokernel handles virtual memory mapping and coherency directly. When a program is loaded by *exec*(2), the UNIX keeper builds an address space segment and copies the executable file segment into it. Manipulating the KeyKOS segment structures is simpler than the equivalent structure manipulations in UNIX, and allows the UNIX keeper to be largely platform independent. The nanokernel is responsible for the construction of mapping tables for the particular hardware platform.

The KeyNIX File and Device System

The UNIX Keeper holds a key to the root inode of the KeyNIX file system. Each inode contains the usual UNIX inode information, and is implemented by a KeyKOS domain. If the inode denotes a file,

the inode domain holds a key to a KeyKOS segment containing the file data. If the inode denotes a device, the device major and minor numbers are contained in the inode.

By making each UNIX inode into a KeyKOS domain, the UNIX Keeper does not have to manage an inode cache or worry about doing I/O to read and write inodes. When the Keeper needs to read the status information from an inode it sends a message to the Inode object and waits for the reply. Similar arguments apply to other operations. The Keeper does not cache file or directory blocks, and does not maintain paging tables for support of virtual memory. All of these functions are handled by the nanokernel.

In the original KeyNIX implementation, directory inodes contained a key to a B-tree domain that was an underlying KeyKOS tool. An analysis of typical directory sizes led to the conclusion that it would be more space efficient to implement small directories (less than five entries) in the inode itself. As a result, directory protocol requests are implemented directly by the inode domain. If the inode does not denote a directory it fails the directory messages appropriately. A curious artifact of this approach is that directory order is alphabetical order. This is occasionally visible to end users as a change of behavior in programs that search directories without sorting them.

When opening a file, the UNIX Keeper issues a message to the file system root inode domain. This domain in turn calls on other domains, until ultimately the request is resolved to a segment key that holds the file content. Once the file has been located, the UNIX keeper maps the segment into the keeper address space and adds an entry to the open file table. The open file table is shared by all UNIX Keepers, and is used to hold dynamically changing information such as the file's current size and last modification date.

When opening a device, the UNIX Keeper receives the major and minor device number from the appropriate inode domain. The major number is in turn handed to the device table domain, which returns a key to the domain that implements the driver. Drivers implemented in the prototype include character I/O, graphics console (supports the X Window System), the null device, sockets, kmem, and the mouse. Support for /dev/kmem is limited to forging those responses necessary to run the *ps*(1) command. In most cases, the device driver domain consists of the original UNIX device driver code linked with a support library that maps the UNIX driver-kernel interface onto KeyKOS key invocations.

The Problem of Signals

The most difficult part of the KeyNIX implementation, was support for the *signal*(2) mechanism. One of the deliberate design decisions of KeyKOS is that domains are single threaded. A domain is either waiting for a message, waiting for a reply to a message, or processing a message. There is no mechanism for stacking messages. This decision increases the reliability of the KeyKOS system, but occasionally requires that queuing domains be inserted into an otherwise straightforward remote procedure call.

UNIX signals are asynchronous with respect to the receiving process. As a result, the implementation of the signal mechanism is one of the more complicated and pervasive (not to say perverse) aspects of the UNIX kernel.[3]

To ensure that the UNIX Keeper is always able to receive signal notifications promptly, trivial queuing domains are required where an operation might block or complete slowly. The purpose of these domains is to queue messages to devices such as ttys and pipes that might otherwise delay the receipt of signals by the UNIX Keeper. The UNIX Keeper delivers these messages through the queue domain, and waits asynchronously for the queue domain to send a message indicating completion of the requested service.

---

3 This is also a significant problem for debugging interfaces, such as */proc*(4) and *ptrace*(2).

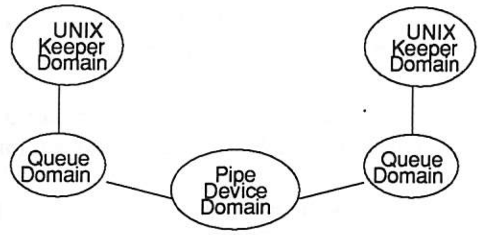


Figure 2: Domains in a Pipe

In effect, a series of fork messages are used to implement a non-blocking remote procedure call to the device domain in order to ensure that the UNIX kernel is always ready to receive another message.

The queue insertion approach has unfortunate consequences for slow devices (with disk devices one can reasonably assume instant service and duck the issue), and severly impacted communication facilities such as pipes or sockets, as shown in Figure 2.

These mechanisms are penalized by the requirement from both sides to remain able to receive signals while proceeding with the I/O transfer. The impact is easily visible in the performance of KeyNIX pipes. A better alternative is discussed below.

Expected Performance

To the best of our knowledge, the KeyNIX system uses far more processes than any other microkernel-based UNIX implementation. Reactions to the KeyNIX design from UNIX developers range from shocked to appalled at the profligate use of processes. UNIX developers find it difficult to accept that the task switch cost can be lower than the data management code that it replaces. We find this ironic, as one of the major innovations of the UNIX system was the notion that processes were cheap.

The object paradigm was at the heart of the design of the KeyKOS system and, as a result, the task switch costs are very much lower than in traditional systems and several times lower than in competing microkernels such as MACH and Chorus. On the Motorola 88x00 series, a typical message send takes less than 500 cycles.[4] The low cost of task switches makes it possible to obtain better performance with much simpler software by taking an object-oriented approach to the decomposition of the system. The UNIX implementation described here takes considerable advantage of KeyKOS building blocks. The complete UNIX kernel implementation is approximately 16,000 lines of C code.

Known Incompatibilities

The KeyNIX implementation is 99% compatible with the Omron BSD 4.3 implementation. While KeyNIX could be equally compatible with MACH 2.5, the existing prototype is not. There are four significant incompatibilities in the prototype:

1. The application prolog ("crt0") in MACH 2.5 initializes certain MACH ports. Because KeyNIX does not yet implement MACH ports, applications built with the MACH 2.5 crt0.o do not run under KeyNIX.

---

4 This time includes the context switch and copying both data and keys. The Motorola implementation is the slowest implementation to date.

2. MACH 2.5 port functions are accessed by a trap instruction in the same fashion as are UNIX system calls. KeyNIX does not implement these traps.

3. In MACH 2.5, the *fork*(2) system call does the same port initialization for the new task that was done by "crt0" in the parent task. This change is not implemented in KeyNIX.

4. MACH 2.5 does not implement the *sbrk*(2) system call. This call is handled by a library routine that uses the "VMALLOC" of MACH 2.5 to handle memory expansion and contraction.

5. The KeyNIX text segment is writable, which can impact buggy programs. This is the result of a quick and dirty implementation, and could be easily fixed.

Programs compiled on the Luna 88K under MACH 2.5 that are to be run in the KeyNIX system must be linked with a new prolog and new library stubs for *fork*(2) and *sbrk*(2). In cases where the ".o" files exist, there is no need to recompile the programs, but the programs must be relinked.

The existing prototype does not support all BSD 4.3 system calls. The major criterion for choosing what to implement and what not to implement was the need to run X-Windows, *csh*(1), *ls*(1) and similar useful utilities. If the system call is not needed to run these applications then it is not implemented. There are a number of calls that are implemented in a limited fashion, again sufficiently to run the required applications. As an example, *csh*(1) makes *usage*(2) calls but does not depend on the answers for correct behavior. *Usage*(2) always returns the same fixed values and is not useful as a measuring tool as a result.

To get an intuitive sense of the compatibility achieved, it may suffice to say that all of the application binaries running on KeyKOS were obtained by copying the binary file from the existing BSD 4.3 system. The X Window System, compilers, shells, file system utilities, etc. all run without change under KeyNIX.

10. Performance Comparison

A limited performance comparison was made between the KeyNIX prototype and the Omron MACH 2.5 implementation. A more careful analysis would be required for any serious evaluation of the two systems for production use. KeyNIX got mixed results for common system call sequences:

Operation	Iterations	KeyNIX	MACH 2.5	Ratio
getpid();	10,000	12,000/sec	30,000/sec	0.4
open();close();	1,000	714/sec	2777/sec	0.26
fork();exit();	100	64/sec	10/sec	6.4
exec();	100	151/sec	12/sec	11.6
sbrk(4096);sbrk(-4096)	100	2564/sec	181/sec	14

I/O performance was equally mixed:

Operation	KeyNIX	MACH 2.5	Ratio
Pipe (round trip)	.588 Mbyte/sec	1.05 Mbyte/sec	.56
Disk access program	4 seconds	26 seconds	6.5

As anticipated, the simplification achieved by adding domains doesn't always lead to better performance. The cases that the KeyNIX prototype handled poorly have straightforward corrections which are discussed below.

Simple System Calls

Simple system calls include calls such as *getprocid*(2), *putprocid*(2), and *gettimeofday*(2), which are essentially accessor functions. A trap is taken, but the system call itself performs little or no interesting activity within the kernel. The KeyNIX system is binary compatible with this approach.

The MACH 2.5 implementation is able to execute these system calls 2.5 times as fast as the KeyNIX system because no context switch is involved. MACH 3 uses special system call libraries to implement some of these functions in the UNIX process address space. A similar approach would be possible in KeyNIX if the system calls were implemented in dynamic libraries, as in System V Release 4, or if binary compatibility could be sacrificed. We were surprised that KeyNIX did so well on this comparison.

Open and Close

To explore the limits of domain performance, we elected to implement each inode as an individual domain. On the basis of our previous experiences, it seemed likely that the simplification achieved by this approach would overcome the overhead of multiple domains. With the benefit of hindsight, we were mistaken, and the performance of *open*(2) suffered excessively. The namei() routine within the UNIX kernel is heavily used, and the decision to use multiple domains in effect inserted four context switches into the inner loop(for two round-trip RPC's).[5] In a small program that simply opens and closes a single file 1,000 times, the MACH 2.5 system outperformed the KeyNIX system by nearly four to one (3.89). Alternative implementations are discussed below.

Fork and Exit

Because the UNIX programming model assumes that processes are cheap, the performance of *fork*(2) is critical to the overall performance of the system. In KeyKOS, the equivalent to *fork*(2) is even more critical, and is possibly the most carefully optimized path in the nanokernel. We therefore expected KeyNIX to do well on *fork*(2) calls. KeyNIX outperforms MACH 2.5 by a little more than six to one.

The current KeyNIX implementation suffers from an extremely naive loader implementation in the UNIX keeper. When performing a *fork*(2), a complete copy of the process address space is made. The implementation could be improved by sharing the read-only text pages rather than copying their content. In addition, it would not be difficult to implement UNIX copy-on-write semantics as part of the segment keeper that services faults on the UNIX address space. Neither of these optimizations was performed in the prototype due to time constraints, and we would expect each to result in substantial improvements.

Exec

Given the naive loader implementation, we were pleasantly surprised to find that KeyNIX outperformed MACH 2.5 by better than eleven to one on *exec*(2) calls. The test program simply calls *exec*(2) one hundred times and exits. Implementing shared text would significantly improve the KeyNIX results.

Sbrk

In order to compare the performance of the *sbrk*(2) system call, a program was written to repeatedly grow and shrink the heap. 100 calls to sbrk(4096) and sbrk(-4096) were executed with a fetch of a byte from the newly allocated memory. The fetch of the byte forces the UNIX implementation to actually allocate

---

5 One round trip to access the inode domain, the second to access the directory domain.

the main store for the page, and consequently forces the page to be deallocated when the heap segment size is reduced. KeyNIX outperformed the MACH 2.5 implementation by fourteen to one, which was consistent with our expectation.

Pipe Bandwidth

Pipe performance is one of the areas where we expected KeyNIX to suffer. In order to compare the pipe implementations, a megabyte of data was passed through a pipe to a child process task and back in 1000 byte chunks. The MACH 2.5 implementation outperformed KeyNIX by nearly two to one.

This result is principally due to the insertion of queue domains into both ends of the pipe, imposing considerable context switch overhead. In retrospect, we could have eliminated the queues and depended on the fact that asynchronous signal delivery timing is not guaranteed by the UNIX process model. In particular, correct UNIX programs cannot depend on the fact that *interprocess* signals will interrupt a system call in the receiving process. Taking advantage of this loophole would allow for a much simpler and faster implementation.

Disk File I/O

To measure disk performance, we built a program to create a large test file and read it repeatedly. The I/O model of KeyNIX and MACH 2.5 are so radically different that other comparisons are very difficult. Uncached writes, for example, are dominated by disk arm movement, so a comparison of such activity is unenlightening. The times reported are the elapsed time to write and then read a one megabyte file ten times. KeyNIX outperforms MACH 2.5 by better than six to one.

KeyNIX I/O performance is a direct result of the underlying KeyKOS I/O design. KeyKOS never writes to disk as a direct result of writing to a file. All writes to the disk are part of the paging, checkpoint, and migration system.

To determine the impact of the checkpoint process on the test, we arranged for KeyKOS to perform a checkpoint and migration in parallel. This process increases the KeyKOS time to 4.4 seconds, giving a performance ratio of 5.9 to one. To the best of our knowledge, the prototype KeyNIX system achieves the highest I/O bandwidth utilization of any UNIX system today.[6] KeyKOS's I/O performance makes the overall performance of many applications better under KeyNIX than under a more conventional system, and appears to more than balance the prototype's performance deficiencies.

Performance Summary

The overall performance of the KeyNIX system is quite comparable with MACH 2.5. Some operations are slower and some quite a bit faster. A user using X-Windows doing VI and using a variety of shell commands and scripts is unaware of any significant performance difference between MACH 2.5 and KeyNIX.

---

6 We are well aware of the significance of the I/O subsystem design in this claim, and believe that the claim would hold up when examined with other I/O subsystems and bus architectures. On the System/370, KeyKOS achieves channel utilization of better than 95% on all channels. With current SCSI technology, KeyKOS's disk utilization is limited by the SCSI channel performance.

11. Implementation Alternatives

In the course of the prototype effort, we came up with several ways to simplify the UNIX keeper and to cut down on some of the overhead. Each of these ideas represents a compromise in the use of domains and multiple instantiation.

Domains for Process and File Table Manipulation

The current process table segment is an array of process table entries. The UNIX process id is used to index the table. Process numbers are reallocated quickly, which leads to certain problems in the human interface for system maintenance. Also there are circumstances when process table entries should be chained so that children can be located more quickly. This is best handled by introducing a domain for process table entry manipulation that allocates and chains process table entries. The UNIX keeper continues to reference its own process table entry directly, but accesses other process table entries (to obtain a signal key) using the process table management domain. Similarly, the open file table could be implemented by a domain. These modifications would both simplify the UNIX keeper and remove the primary impediment to distribution of the KeyNIX implementation on loosely coupled architectures.

Small Files

The data for small files could be kept in nodes instead of segments. A small file might be a single-level tree of nodes with up to 16 leaf nodes each holding 176 bytes of data. When the 17th node is required the file is converted to a segment. The inode domain would convert the file to a segment when it is opened, and on the last close would convert it back into node form if it is small enough. This would allow KeyNIX to achieve more efficient storage of small files than current UNIX systems.

File System Domain

Opening files is a crucial operation in UNIX systems, and the domain-per-inode approach is not nearly fast enough. Two alternative implementations would have delivered competitive performance.

The first approach is to build the entire directory and inode support structure for a file system into a single domain, while continuing to implement files as individual segments. This would eliminate almost all of the context switching performed in the file subsystem, and would probably outperform the MACH 2.5 implementation.

The second alternative is to implement a compatibility library that would enable us to simply compile a vnodes-compatible file system into a domain. Using this approach, the entire file system would reside in a single KeyKOS segment, and bug-for-bug compatibility is achievable. This approach is something like the File Manager tasks of CHORUS and MACH 3. In practice, supporting vnodes file systems is probably a compatibility requirement for a commercial UNIX implementation, but system reliability suffers greatly from this requirement.

Our current preference would be the first alternative, mainly to eliminate the bugs of the existing file system implementations. In addition, we feel that this approach significantly simplifies recovery in the event of a disk block failure, as it eliminates the need for a complicated file system consistency checker.

12. Conclusions

The KeyKOS nanokernel has been running in production environments for nine years. It is proven technology, and we feel that the architecture and implementation have much to offer to the computing community at large. A serious development project could far exceed the performance that we obtained from the six month UNIX prototype effort.

KeyKOS represents a pardigmatic shift in operating system technology. It is therefore difficult to make direct comparisons with other approaches. A pure capability architecture brings fundamentally greater discipline, control, and reliability to application construction. In the long term, we feel that this degree of reliability is necessary to realize the productivity promises of the information age.

For further information on KeyKOS:

U.S. Mail: Norman Hardy
143 Ramona Road
Portola Valley, CA 94028

Phone: (415) 851-2582

Email: norm@xanadu.com

13. Bibliography

1. Theodore A. Linden, "Operating System Structures to Support Security and Reliable Software," NBS Technical Note 919, U.S. Department of Commerce, National Bureau of Standards, Institute for Computer Sciences and Technology, August 1976. (Also published in ACM *Computing Surveys*, 8, 4, December 1976, pp. 409-445).

2. Norman Hardy, "The Keykos Architecture," Operating Systems Review, September, 1985.

3. *Introduction to KeyKOS Concepts*, KL004, Key Logic, 1988.

4. *KeyKOS/370 Principles of Operation*, KL002, Key Logic, 1988.

5. *KeyKOS Architecture*, KL028, Key Logic, 1988.

6. Butler Lampson, "A Note on the Confinement Problem," *Communications of the ACM*, 16, 10, October 1973.

7. Henry M. Levy, *Capability Based Computer Systems*, Digital Press, 1984.

8. M. Ritchie and K. L. Thompson, "The UNIX Time-sharing System." *Communications of the ACM*, July, 1974.

9. *System/370 Principles of Operation*, GA22-7000-9, IBM, 1983.

10. Patent number 4,584,639 (describes the secure factory mechanism).

11. William A. Wulf, Roy Levin, and Samuel P. Harbison, *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill Book Company, 1981.

An Architectural Overview of QNX

Dan Hildebrand
Quantum Software Systems, Ltd.
175 Terrence Matthews
Kanata, Ontario K2M 1W8
Canada
(613) 591-0931

danh@quantum.on.ca

Abstract

This paper presents an architectural overview of the QNX operating system. QNX is an OS that provides applications with a fully network- and multiprocessor-distributed, realtime environment that delivers nearly the full, device-level performance of the underlying hardware. The OS architecture used to deliver this operating environment is that of a realtime microkernel surrounded by a collection of optional processes that provide POSIX- and UNIX-compatible system services. By including or excluding various resource managers at runtime, QNX can be scaled down for ROM-based embedded systems, or scaled up to encompass hundreds of processors—either tightly or loosely connected by various LAN technologies. Conformance to POSIX standard 1003.1, draft standard 1003.2 (shell and utilities) and draft standard 1003.4 (realtime) is maintained transparently throughout the distributed environment.

Architecture: Past and Present

From its creation in 1982, the QNX architecture has been fundamentally similar to its current form—that of a very small microkernel (approximately 10K at that time) surrounded by a team of cooperating processes which provide higher-level OS services. To date, QNX has been used in nearly 200,000 systems, predominantly in applications where realtime performance, development flexibility, and network flexibility have been fundamental requirements. The large installed base has proven that microkernel technology is both commercially viable and suitable for mission-critical applications such as process control, medical instrumentation, and financial transaction processing. The performance needs of these applications has been a significant driving force in the evolution of QNX from version 1.00 up through version 3.15.

In 1989, the development of a POSIX-compliant version of QNX (QNX4.0) began with the goals of maximizing the performance and flexibility delivered by the previous generation of the product. This new version was released in 1991. This paper will detail the features of the new architecture and discuss its strengths and limitations, as well as areas targeted for future development.

A True Microkernel

The QNX microkernel implements four services: interprocess communication, low-level network communication, process scheduling, and interrupt dispatching. There are 14 kernel calls associated with these services. In total, these functions occupy roughly 7K of code and provide the functionality and performance of a realtime executive (Appendix A). Given the small kernel size, processors that provide a reasonable amount of on-chip cache can deliver excellent performance for applications that heavily use the services of the microkernel, since the microkernel and the system interrupt handlers can often fit comfortably within an on-chip CPU cache of 8K.

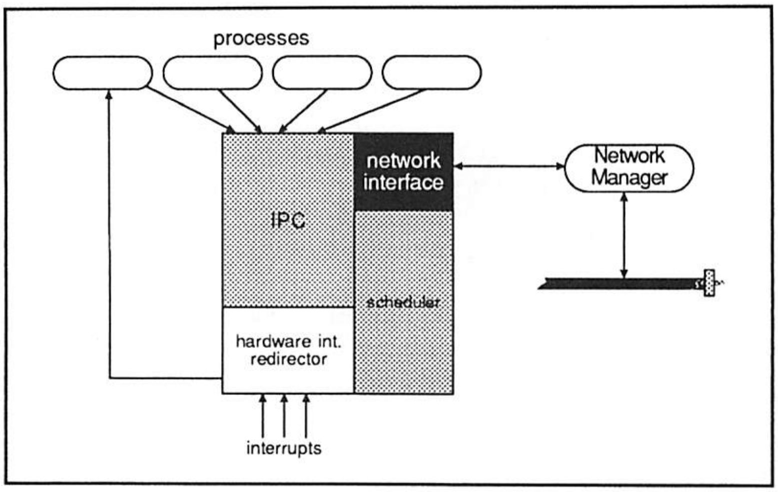


Figure 1. — The QNX Microkernel

The message passing facilities provided by the microkernel are blocking versions of Send, Receive, and Reply. To summarize, a process that does a Send() to another process will be blocked until the target process does a Receive(), processes the message, and does a Reply(). If a process executes a Receive() without a message pending, it will block until another process executes a Send(). Since these primitives copy directly from process to process without queuing, message delivery performance approaches the memory bandwidth the underlying hardware. All system services are built upon these message-passing primitives. Variations on these IPC primitives (e.g. message queues) have been easily implemented as servers employing these lower-level services. Performance of these alternate IPC servers is comparable and often superior to the performance of these services implemented within monolithic kernels.

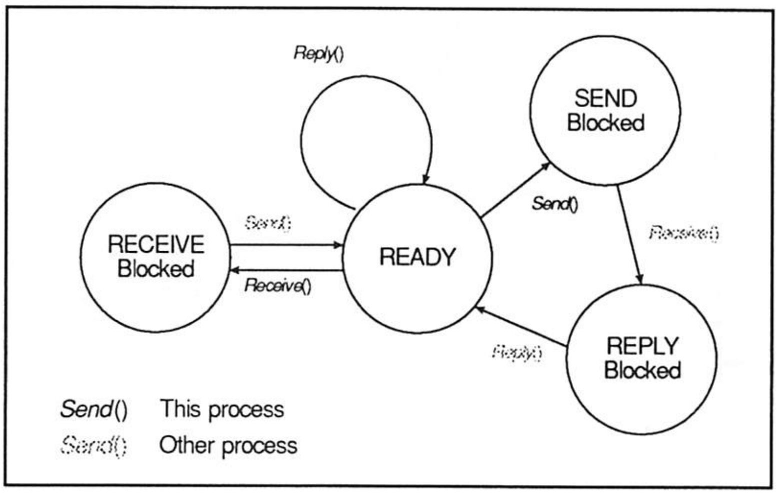


Figure 2.

Processes can request that messages be delivered in priority order (rather than in time order) and that process execution proceed at the priority of the highest-priority blocked process waiting for service. This message-driven priority mechanism neatly avoids the priority inversion problems that can result in fixed-priority message passing systems. Server processes are forced to execute at the priority of the process they are serving,

and yet automatically have their priority appropriately boosted when a higher priority process blocks on the busy server. As a result, a low priority process cannot preempt a higher-priority process by invoking the services of an even higher-priority server.

The messaging primitives support multi-part messaging, such that a message delivered from one process to another need not occupy a single, contiguous area in memory. Instead, both the sending and receiving processes can specify an MX table that indicates where the sending and receiving message fragments exist in memory. This allows messages that have a header block separate from the data block to be sent without performance-consuming copying of the data to create a contiguous message. In addition, if the underlying data structure is a ring buffer, a three part message will allow a header and two disjoint ranges within the ring buffer to be sent as a single, atomic message. The MX mapping applied to the message by the sender and the receiver need not be the same.

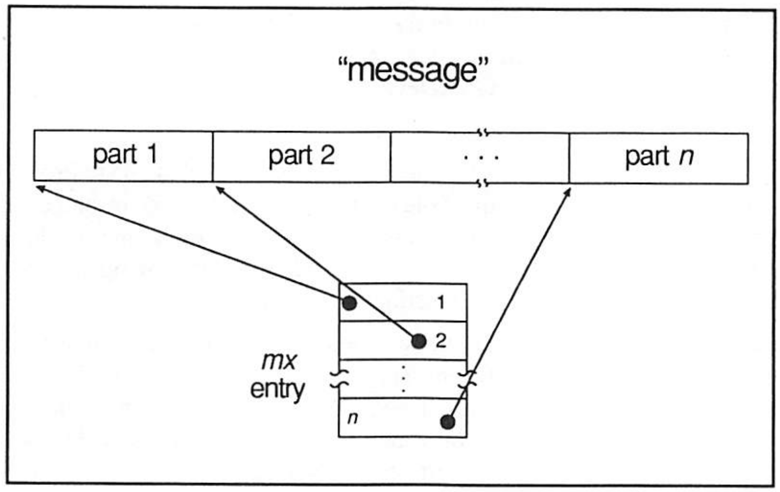


Figure 3.

Low-level network communications are designed directly into the microkernel and are provided by an optional process known as the Network Manager (described later in this paper). When present, the Network Manager is directly connected into the microkernel and provides the microkernel with the facilities needed to move messages to and from other microkernels on the LAN. By providing network services at this fundamental level in the system, any services provided in higher architectural layers of the operating system are transparently accessible to any process, anywhere on the network. Architecturally speaking, this implementation is very lean, providing as efficient an interface as possible.

The process-scheduling primitives provided by QNX conform to the POSIX 1003.4 (realtime) draft specification. Fully preemptive, prioritized context switching with round-robin, FIFO, and adaptive scheduling are provided. As the POSIX 1003.4 standard emerges from draft status, the microkernel and system processes will be evolved to match.

Resource Managers and Pathname Space Management

For the microkernel to deliver the functionality specified by POSIX standards and UNIX conventions, optional processes known as resource managers can be added. A minimal system, without a filesystem or device I/O system, can be built from a microkernel, a Process Manager, and a set of application tasks.

The first, and only mandatory, resource manager is the Process Manager (Proc). Proc provides services such as process creation, process accounting, memory management, process environment inheritance (both locally and for network remote processes) and pathname space management. First-level pathname management is

done by Proc because, unlike a monolithic-kernel system where the filesystem is always present, a filesystem is optional under QNX. Diskless or ROM-based systems may have no use for a filesystem, and so are not forced to use one.

Until resource managers begin execution, Proc "owns" the entire pathname space (the root and everything beneath it). Without any resource managers present to provide services, this is essentially an empty filesystem. Proc allows resource managers, through a standard API, to adopt a portion of the namespace (a "domain of authority") that they would like to administer. Proc is then responsible for maintaining a prefix tree to track the processes that own various portions of the pathname space.

When Fsys (the filesystem manager) and Dev (the device manager) are running, the prefix tree would look something like this:

/	*Disk-based filesystem (Fsys)*
/dev	*Character device system (Dev)*
/dev/hd0	*Raw disk volume (Fsys)*
/dev/null	*Null device (Dev)*

When a process opens a file, the open() library routine first sends the filename to Proc where the pathname is applied against the prefix tree in order to direct the open() to the appropriate resource manager. In the case of partially overlapping regions of authority, the longest match to the pathname would win. For example, if /dev/tty0 were opened, the longest match would occur on /dev, causing the open to be directed to Dev. The pathname /usr/fred would match against '/', directing the open to Fsys.

The Process manager on each computer in the network maintains its own prefix tree and may present identical or different views of the network-wide pathname space to processes on each node. Pathnames that start with a '/' are applied against the prefix tree on that node. Network-unique names are also available to allow applications to specify the absolute location of resources within the network-wide pathname space. Through the use of prefix aliasing, portions of the namespace can be mapped to resource managers on other network nodes. For example, a diskless workstation that booted from the LAN and wished to have its filesystem root on another node could alias the root of its filesystem to a remote Fsys process.

With this alias in place, open() calls to /dev would still map to the local Dev process for control of local devices, but all open() calls for files would result in open messages being resolved by the prefix mapping table on the previously specified remote node (which would usually direct file opens to the Fsys process on that node). As a result, processes anywhere on the network can access all of the network filesystem resources within a single directory tree, connected to a common root. Alternatively, by using network absolute pathnames, the network pathname space can also be manipulated as a collection of individual root filesystems.

By implementing individual "domains of authority" within the conventional filename space, portions of the overall functionality of the OS can be implemented in a runtime-optional manner. Since resource managers live outside the kernel space and are not fundamentally different from user processes, they can be added or removed dynamically, at runtime, without requiring that the kernel be relinked to contain different levels of functionality. This flexibility in sizing allows the OS to be easily scaled up or down, depending on application needs.

Although placing the services provided by resource managers outside the kernel would at first appear to be inefficient, the performance results given in Appendix B indicate that the context switch and IPC performance of the microkernel are more than adequate to keep up with the raw performance of the hardware.

The network transparency of this namespace allows remote execution of processes to be logically equivalent to execution on a local processor. The individually administered pathname spaces blend seamlessly and there are no "surprises" in how the namespace behaves. Inheritance of the entire parent process environment, including open file descriptors, environment variables, and the current working directory is done such that interprocessor communications and file I/O operate in accordance with the POSIX 1003.1 specification in spite of the network-distributed runtime context.

Fsys—The Filesystem Manager

Fsys is the resource manager that provides a POSIX-compliant filesystem for the QNX environment. It implements a disk structure that uses a bitmap for free space allocation, and a linked-list-of-extents approach to organizing the data on disk. This approach allows the system to deliver disk throughput at the application level that approaches the raw capacity of the hardware (Appendix B). Fsys performs synchronous writes to disk for data structures critical to filesystem integrity, allowing the disk system to gracefully survive unexpected power outages. Special tags embedded in on-disk data structures allow the filesystem to be easily rebuilt in the event of catastrophic failures as well.

The multithreaded architecture of Fsys allows it to deal with multiple requests in parallel, such that ramdisk and cache I/O can occur while other threads are blocked, waiting for physical I/O to occur. This parallelism extends down into the driver as well, such that if a device can support multiple pending I/O requests, they can be serviced by the drive in whatever order is appropriate.

Although an initial study of message-passing operating systems might suggest that a filesystem would need to copy data around more so than a monolothic kernel filesystem, the reality is that no additional copying is needed. The MX multipart messaging primitives allow Fsys to map the contiguous buffers specified by the read() and write() calls of the application into the non-contiguous cache blocks within Fsys. For a disk read, the disk driver reads from disk into multiple non-contiguous, LRU-allocated cache blocks. Fsys then invokes the MX facility within the kernel to atomically gather and copy the scattered blocks into the contiguous read buffer specified by the application. As a result, even though the filesystem exists within a message-passing, network-transparent environment, it exhibits the same amount of data copying that would occur with a filesystem implemented in a monolithic kernel.

The Fsys process can be started from the command line on a diskless, network-connected machine and device drivers can then be dynamically attached to Fsys. In the event that Fsys is no longer required, Fsys and its drivers can be removed from memory.

Dev—The Device Manager

The Device Manager (Dev) provides POSIX-compliant device control with some extensions suitable for realtime communications. In a manner similar to Fsys, it can be dynamically started and its device drivers attached and then later removed from memory if no longer needed.

Dev can handle baud rates up to 115 Kbaud on modest hardware with non-intelligent UART devices because of the low interrupt latency provided by the microkernel. With the addition of intelligent communication boards, a high bandwidth, multiline communications server can be configured.

Use of the MX messaging primitives allows Dev to Receive() the write() done by an application to a device directly into a ring buffer managed by an interrupt handler. With the MX table appropriately defined, the data received can be laid directly into the ring buffer managed by Dev. Since writing to a ring buffer can require that the data be mapped into physically disjoint (but logically contiguous) memory regions, an MX table with three entries can describe the header and the two physically disjoint sections of the ring buffer. For the read() case, the data flow from a device goes directly from the driver into the ring buffer, and from the ring buffer into the application's read() buffer without redundant copying to build contiguous messages.

Device Driver Support

Rather than insisting that device-driver interrupt handlers live only in the kernel space, a system call is provided that allows user processes to connect a handler within a sufficiently privileged user process to a particular interrupt vector within the kernel. The connected handler can then be called by the kernel in response to physical interrupts. By existing within the user process, the handler has full access to the address space of the process for the purpose of responding to the interrupt. Once the handler has run, it can either wake up the process it shares code with or simply return to the kernel. The device drivers for Dev take advantage of this behavior by using the individual interrupts to accumulate characters within a Dev managed buffer, waking Dev only when a previously defined "significant event" has occurred, such as when a terminal character count, end-of-line condition, or timeout has occurred.

With interrupt handlers existing outside the kernel in this manner, it becomes possible for interrupt handlers (and the device drivers that contain them) to be dynamically added and removed from a running system. The first-level interrupt handling done by the kernel also takes care of nested and shared interrupts without imposing the hardware-dependant details and complexities on user-written interrupt handlers. External interrupt handler support for the microkernel is fundamental to allowing a resource manager to match the level of performance that resource management within a monolithic kernel could provide.

Ease of Extension:

A fundamental advantage to having device drivers exist within user-level processes is that developing extensions to the OS is not functionally different from developing user-level processes. In fact, the development approach used in-house is to execute experimental resource managers under the control of the full-screen, source-level debugger, so that debugging OS services like a new Fsys process can be done without having to resort to low functionality tools such as the kernel debuggers typically used to debug kernel-linked extensions for monolithic kernel operating systems. Since resource managers and device drivers can be started and removed at will, the laborious process of relinking a kernel and rebooting to test the new kernel becomes entirely unnecessary.

As an example of how easily extensible the system is, services such as a /proc resource manager similar to that described in [Pike90] have been implemented by applications-level programmers (not kernel architects!) with only a few hours of effort and less than 200 lines of easily understood C source. In effect, the /proc resource manager packages up a system resource (the list of active processes in the system) and then presents it to the system as files and directories that can be manipulated within /proc pathname space.

As a more complex example, a client-side network filesystem cache manager similar to that described by [Presotto91] has been implemented with approximately a week of effort. This cache keeps copies of recently accessed file blocks at the client-side for a node accessing a network remote Fsys. At open(), it verifies that the remote file has not changed (which would invalidate the locally cached data) and provides the locally cached data as a performance enhancement. By providing a local disk file for this client-side cache to "spill" into, it becomes reasonable for a system networked over a slow serial link to still provide reasonable network-remote filesystem performance. This server represented only 1000 lines of source and again, was fully within the reach of an application-level programmer using standard system libraries.

Finally, guest filesystems can be implemented as a resource manager that uses the raw disk block I/O services of Fsys to present a guest filesystem as a subtree within the root filesystem. One example is Dosfsys, a PC-DOS filesystem. Dosfsys adopts the "/dos" pathname as its domain of authority and then presents to the system a series of directories of the form /dos/a, /dos/b, etc. These directories map onto corresponding PC-DOS media and the Dosfsys process manipulates the raw blocks on these volumes as indicated by the I/O requests that enter Dosfsys. File manipulations that can be mapped onto the underlying filesystem are supported, while others—such as link()—return the appropriate error status when attempted.

Network Services—FLEET™ networking technology

F-ault Tolerant
L-oad balancing
E-fficient
E-xpandable
T-ransparent networking

As mentioned previously, the Network Manager (Net) is directly connected into the microkernel. When the microkernel is invoked to pass a message from a local process to a process on another node, it enqueues a pointer to the message through this private interface to Net. Similarly, Net can receive messages from other microkernels and give those messages to the local microkernel. Essentially, the network managers on the network merge the many remote microkernels into a single microkernel. Since all system services, including process creation, debugging, file and device I/O are accomplished via message passing through the microkernel, the result is a network of machines that behave like a single computer. Any services provided in higher architectural layers of the operating system are transparently accessible to all processes on the network. This is in marked contrast to a TCP/IP services suite, which provides only very explicit sets of services—typically terminal sessions and file access. By comparison, this connected microkernel architecture allows the command:

ls /usr/danh | grep abc | wc

to be run such that every process will run on a different processor on the network, while the network-inherited file descriptors provided by Proc cause the pipes to connect and forward data over the network. The transparency of this environment also facilitates the implementation of distributed applications. For example, the development of a network-distributed team make utility was accomplished with only a man-week of effort, starting from a conventional, non-distributed make.

Just as Fsys and Dev can be started and stopped from the command line, each having a family of drivers, Net has a family of drivers and supports the connection of multiple network drivers to Net. If Net discovers that more than one of the net drivers provides connectivity to the same node, it will load balance the traffic between the drivers. The load balancing uses an algorithm based on media transmission rate and queue depth. Command-line options are available to manually coerce network traffic as desired.

Use of multiple network paths between nodes provides better throughput and fault tolerance by adding extra network links between network nodes. Application-level changes are not needed to take advantage of this fault tolerance, since the support exists locally within the Network Manager.

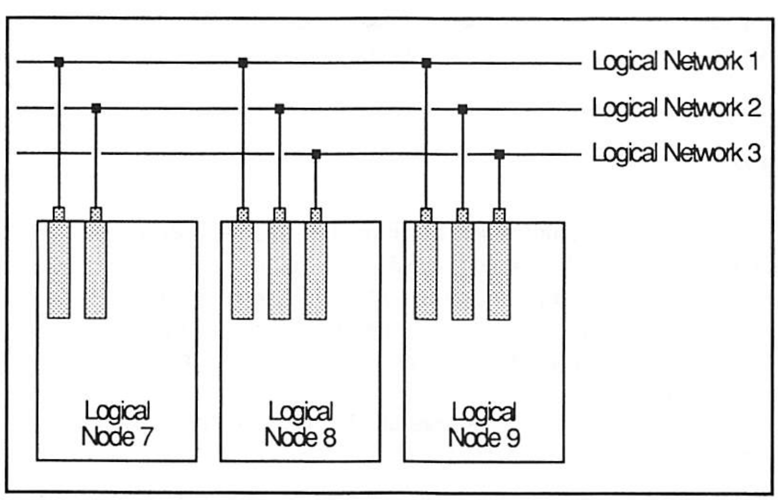


Figure 4.

Inexpensive serial links can be used as a fall-back network link in case the main LAN fails. By running the serial link at a high baud rate (Dev is capable of 115 Kbaud), by doing data compression, and by enabling client-side filesystem caching, serial network performance can be very snappy.

This facility may also be used to resolve the LAN congestion problems that result when two file servers on a LAN experience a high volume of point-to-point traffic. With the FLEETnet approach, a private network link can be added to join the two servers, moving the point-to-point traffic off the main LAN and onto the private link. If the two servers are physically adjacent, unconventional LAN technologies such as point-to-point SCSI or bus-to-bus DMA become viable options. This approach can be used to implement CPU/Fileserver groups much as described in [Presotto91].

The FLEETnet approach also allows a system backplane bus to be used to construct a multiprocessor system by building a processor board that uses the backplane bus as a VLAN (Very Local Area Network). Each processor board would run a QNX OS consisting of a microkernel, Proc, Net, and a Net.vlan driver. One of the processors could run Net with both a Net.vlan driver and a Net.ethernet driver to access an external ethernet LAN. By adding additional hardware to these processors, and the appropriate Dev or Fsys processes, they become distributed I/O processors. Currently, a test implementation of this architecture using a Microchannel bus for the VLAN is under joint development with AOX Incorporated. The Microchannel burst mode and multimaster bus arbitration will perform very well as a VLAN. In effect, each node on an ethernet could contain a VLAN of additional processors within the chassis. This lets the team of processes that normally run on each ethernet node to redistribute and run on a team of processors within that node. The compute servers described by [Tanenbaum89] can be readily implemented with this hardware.

For embedded applications, a minimal QNX system can be put into less than 100K of ROM (microkernel, Proc, and some applications). With the addition of a Net task and a Net driver (approx. 35K), the embedded system could then be connected to a larger network, becoming a seamless extension of the larger LAN. This would allow the embedded system to access databases, graphical user interfaces, LAN gateways, and other services. In spite of the limited functionality of the embedded system, the network link out to the LAN provides access to the entire LAN's resources for the processes running on the embedded system. The embedded system could also boot from the LAN, further reducing its ROM requirements. Because the system debugging services are implemented through standard messages to the Proc process on the node running the application being debugged, applications on the embedded system can be debugged from any other node on the LAN.

To host standard transport protocols, a Clarkson-compatible raw packet delivery service provided by Net and the network drivers is available. With a protocol stack implemented in this manner, non-QNX machines on the same physical LAN can communicate through the protocol stack to access the services of the multi-processor QNX LAN. The QNX environment would appear to the outside world (e.g. TCP/IP) as a single, multiprocessor machine.

Currently, FLEETnet does not support network bridging. This requires that communication between nodes not connected to a common network will need to make use of intermediate agent processes to pass messages from LAN to LAN. Research is in progress to define a routing process running as an adjunct to Net to perform this function.

Maintainability:

A fundamental problem with the maintenance of a monolithic kernel operating system is that all of the kernel code runs in a common, shared address space. The danger that one portion of the kernel might corrupt the data space of another is very real, and must be considered every time new drivers are linked into the kernel. The approach taken by QNX is to explicitly define the interfaces between the components that make up the OS, such that each resource manager, just like user processes, run in its own memory-protected space, and all communications between the OS modules is through standard system IPC services. As a result, errors introduced by one resource manager will be constrained to that subsystem and will not corrupt other, unrelated resource managers in the system.

Given that new resource managers and device drivers can be debugged and profiled using the same tools as would be used on user processes, system development becomes at least as well instrumented as application development. This is very important, as it allows much greater freedom to experiment with new approaches to implementing OS subsystems without incurring the tremendous effort of debugging a kernel with limited tools.

The architecture also demonstrates a relative simplicity and ease of implementation that allows the maintenance of existing code to be manageable, and the addition of new features to be a task with a reasonable scope. The following table presents the source line count and code size for the various modules that make up the system (all of the source line counts in this paper were generated by counting the semicolons in the C source files).

	Lines of Source	Code Size
Microkernel	605	7K
Proc	3924	52K
Fsys	4457	57K
Fsys.ahascsi	596	11K
Dev	2204	23K
Dev.con	1885	19K
Net	1142	18K
Net.ether	1117	17K
	15930 lines	204 Kbytes

This source line count compares favorably to [Pike90], although the design goals for the two systems are somewhat different.

Future directions:

Now that QNX is UNIX source code compatible, development of binary compatibility is underway. The combination of source and binary compatibility (ABI) will allow existing UNIX applications to be hosted on a QNX runtime platform and benefit from network-transparent distributed processing and enhanced system performance.

The emergence of commercially successful symmetric shared memory, multiprocessor machines has also raised the issue of multiprocessor support in the QNX microkernel. Given that the microkernel is less than 7K in size, the complexities of multiprocessor support can be constrained to a well defined portion of the system, and will result in a robust implementation. Since the resource manager processes that provide the remainder of the operating system services are multithreaded, independent processes, they will inherit the multiprocessor support provided by the microkernel without modification and the individual components of the operating system will then achieve true concurrency.

Performance:

A necessary challenge that QNX had to meet was the performance needs of a customer base primarily concerned with realtime applications. Although an elegant OS architecture is a joy to work with, "academic elegance" will not necessarily create a commercially successful operating system—it must also provide performance better than traditional monolithic kernel operating systems. A design goal of many of the current microkernel operating systems has been to attempt to match the performance of monolithic kernel systems [Guillemont91]. Given that current monolithic systems, such as UNIX SVR4, fail to deliver the full performance of the hardware (Appendix B), matching only this level of performance will fail to provide consumers

of operating system technology with a significant advantage to using a microkernel-based system. Much as with RISC processors, until a new technology can deliver a clear performance advantage, it will remain little more than an architectural detail to an end-user, and not a factor to influence buying decisions.

For QNX to deliver the full performance of the hardware to the application level (and to exceed the performance of monolithic kernel operating systems), a number of architectural innovations were developed. A necessary precondition for these enhancements was that realtime system performance not be compromised. Even though the increased generality of the message-passing model might at first study indicate that it has more overhead than the monolithic kernel, there are a number of architectural ideas that correct for this situation. Two concepts that contributed significantly to overall system performance was the support for interrupt handlers directly within resource managers, and the multipart messaging primitives.

Appendix B contains the performance results for both a DELL UNIX SVR4 v2.1 implementation and a QNX 4.1 implementation performing similar operations. DELL UNIX was chosen for its reputation as a well-performing port of the SVR4 product to the Intel 80x86 architecture. In the first section the timing for a typical UNIX kernel call (umask) is compared, but under QNX umask() is actually implemented as a message to Proc. The QNX system call rate for umask() is roughly one third that of UNIX, given that these calls represent a different sequence of execution for QNX than for UNIX. The sequence executed is:

1) The test program calls Send() in the kernel
2) The kernel schedules Proc to run
3) A context switch to Proc
4) Proc returns from the Receive() call it was blocked on
5) Proc processes the request
6) Proc calls Reply() in the kernel
7) Kernel schedules test program to run
8) The test program returns from the Send() kernel call

In effect, QNX is doing two kernel calls, doing two message passes, executing the scheduling code twice, and performing two context switches in roughly the time it takes the UNIX system to perform three kernel calls. During this process, there are two points at which other processes can be scheduled, rather than only at system timer intervals. It might appear that an obvious optimization would be to add more kernel calls to the microkernel. However, note that these operations are not those that typically form the bottleneck for system-level performance. Another advantage for these calls to remain as messages to Proc is that they are network-transparent and can be invoked from any processor on the network.

The Yield() call is a true kernel call under QNX, and the results in Appendix B show the kernel call rate to be more than three times that of the UNIX kernel. Implementing the other calls measured in this report in the microkernel would have resulted in much faster kernel call times, but since these are not a performance bottleneck for the system, there is no pressing need to enlarge the kernel to accommodate them. Additionally, the greater the complexity of the microkernel, the slower the more important kernel calls will become, until the microkernel has grown back into a monolithic kernel, with the limitations this implies.

At the system performance level, for IPC, pipe I/O, and disk I/O, we see that QNX outperformed the UNIX system by a substantial margin. In fact, the QNX system was able to deliver virtually all of the raw device throughput to the application, while the SVR4 system fell far short. For disk I/O, QNX was substantially faster than SVR4. As faster peripheral devices appear, the ability to deliver the full performance of that hardware will make possible a class of applications that the kernel overhead of UNIX will not be able to accommodate without much larger investments in processor power. In the network case, the QNX Net process and its drivers deliver very nearly the entire cable bandwidth to the application, even with only moderately powerful machines.

Conclusion

With the experience gained from implementing and analyzing the QNX microkernel architecture, it is clear that a microkernel system can both outperform and provide greater functionality than a monolithic kernel system while still providing a compatible API for application programs. Existing application source code continues to work unchanged, yet the development of OS extensions becomes much easier. The flexibility of the OS platform also paves the way for greater variety and easier experimentation with alternative operating system features as well. Much as innovations in RISC processor architectures have generated a flurry of new performance capabilities in computer hardware, the microkernel OS architecture will generate a renaissance of new performance and functionality standards in operating systems technology.

References:

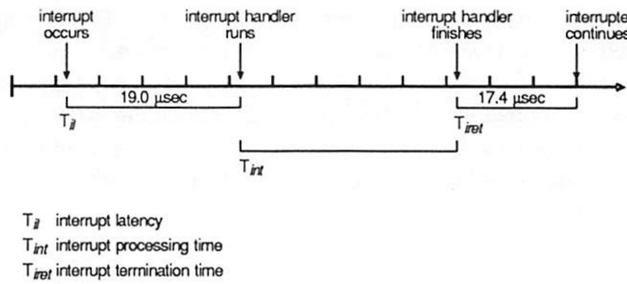
[Guillemont91] M. Guillemont, J. Lipkis, D. Orr and M. Rozier, *"A Second-Generation Micro-Kernel Based UNIX; Lessons in Performance and Compatibility"*, Proc. of the Winter 1991 USENIX Conference, Jan. 1991, pp. 13-22

[Pike90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, *"Plan 9 from Bell Labs"*, Proc. of the Summer 1990 UKUUG Conf., London, July, 1990, pp. 1-9

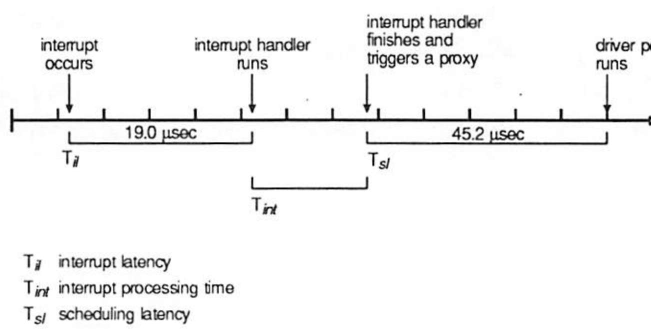
[Presotto91] Dave Presotto, Rob Pike, Ken Thompson, and Howard Trickey, *"Plan 9, A Distributed System"*, Proc. of the Spring 1991 EurOpen Conf., Troms, May, 1991, pp. 43-50

[Tanenbaum89] Andrew Tanenbaum, Rob van Renesse, Hans van Staveren. *"A Retrospective and Evaluation of the Amoeba Distributed Operating System"*. Technical Report, Vrije University, Amsterdam (The Netherlands), October 1989, pp. 27

Appendix A



Interrupt handler simply terminates. Times are for a 20 MHz 386 processor in protected mode.



Interrupt handler terminatesand triggers a proxy. Times are for a 20 MHz 386 processor in protected mode.

The interrupt latency (Til) in the above diagram represents the minimum latency—that which occurs when interrupts were fully enabled at the time the interrupt occurred. Worst-case interrupt latency will be this time plus the longest time in which QNX, or the running QNX process, disables CPU interrupts.

Interrupt and Process Latency

Processor	**Typical Interrupt Latency (T_{il})**	**Interrupt Termination time (T_{iret})**	**Scheduling Latency (T_{sl})**	**Context Switch**
33 Mhz 486	6µsec	5µsec	14µsec	17µsec
25 Mhz 486	8µsec	7µsec	18µsec	22µsec
33 Mhz 386	11µsec	10µsec	27µsec	33µsec
20 Mhz 386	19µsec	17µsec	45µsec	55µsec
16 Mhz 386SX	32µsec	29µsec	77µsec	94µsec
8 Mhz 286	65µsec	59µsec	163µsec	188µsec

Appendix B

System performance numbers comparing QNX4.1 to SVR4 UNIX.

Hardware Environment:

Processor:	Intel 80486 at 33 MHz, ISA bus
Cache:	8K on chip, 0K off chip
RAM:	16 Megabytes
Disk:	1.2 Gigabyte Micropolis SCSI Disk
Controller:	Adaptec 1542B

Software Environment:

A default installation of QNX4.1 with the pipe manager was used for the QNX benchmarks.

A default installation of DELL SVR4 v2.1 UNIX was used for the UNIX benchmarks.

Both QNX and DELL UNIX were run in multiuser mode. The QNX system used a fixed size 2 Megabyte cache and the DELL system used the default SVR4 caching algorithms.

Results:

Kernel Call:

		QNX	UNIX	Ratio
umask	umask() system call (umasks/sec)	10560	28743	0.37
Yield	Yield() system call (yields/sec)	99760	n/a	3.49 ①
message	message passing (msgs/second)	26296	1887	13.94 ②

① Since the Yield() call is defined in POSIX 1003.4 and is not supported under DELL UNIX, we will assume that if it was supported, the UNIX kernel would be able to perform it as quickly as the umask() call. Making this assumption allows us to compute a comparison ratio. The Yield() kernel call under QNX is implemented in a manner roughly comparable to the umask() kernel call in UNIX and serves well for comparison of kernel entry overhead.

② QNX is using its native Send()/Receive()/Reply() messaging primitives while UNIX is using its standard message-passing facilities.

Pipe I/O:

Block size	QNX	UNIX	Ratio
1024 bytes (bytes/second)	1948398	916598	2.13
16384 bytes (bytes/second)	3886920	2114722	1.84

Sequential file I/O:

Write a 16 Mbyte file, and then read it, using 8192 byte read() and write() calls. Both the UNIX UFS and S5-1K filesystems were tested.

	QNX	UNIX UFS	Ratio
Read (bytes/second)	1430404	289811	4.94
Write (bytes/second)	777875	262513	2.96

	QNX	UNIX S5-1K	Ratio
Read (bytes/second)	1430404	175200	8.16
Write (bytes/second)	777875	60068	12.95

QNX Network Throughput:

Measured as the data transfer rate from a user process on one node to a user process on a second node across a private, two node network. Each node is a 33 MHz 386.

Arcnet theoretical maximum:	200800 bytes/second	2.5 Mbits/second
Single Arcnet:	190 K/second	95% efficient
Dual Arcnet:	380 K/second	95% efficient

Ethernet theoretical maximum:	1185840 bytes/second	10 Mbits/second
Ethernet:	960 K/second	81% efficient

Readers familiar with the transfer rates seen with NFS on an ethernet with this class of processor will appreciate these performance numbers.

An Architectural Overview Of The Alpha Real-Time Distributed Kernel

Raymond K. Clark
Concurrent Computer Corp.
rkc@westford.ccur.com
+1 508 392 2740

E. Douglas Jensen
Digital Equipment Corp.
jensen@helix.enet.dec.com
+1 508 493 1201

Franklin D. Reynolds
Open Software Foundation
fdr@osf.org
+1 617 621 8721

Abstract

Alpha is a non-proprietary experimental operating system kernel which extends the real-time domain to encompass distributed applications, such as for telecommunications, factory automation, and defense. Distributed real-time systems are inherently asynchronous, dynamic, and non-deterministic, and yet are nonetheless mission-critical.The increasing complexity and pace of these systems precludes the historical reliance solely on human operators for assuring system dependability under uncertainty. Traditional real-time OS technology is based on attempting to assert or impose determinism of not just the ends but also the means, for centralized low-level sampled-data monitoring and control, with an insufficiency of hardware resources. Conventional distributed OS technology is primarily based on two-party client/server hierarchies for explicit resource sharing in networks of autonomous users. These two technological paradigms are special cases which cannot be combined and scaled up cost-effectively to accommodate distributed real-time systems. Alpha's new paradigm for real-time distributed computing is founded on *best-effort* management of all resources directly with computation completion time constraints which are expressed as *benefit functions*; and multiparty, peer-structured, trans-node computations for cooperative mission management.

1. Introduction

The Alpha OS kernel is part of an multi-institutional applied research and advanced technology development project intended to expand the domain of real-time operating systems from conventional centralized, low-level sampled-data, static subsystems, to encompass distributed, dynamic, mission-level systems.

This paper begins with a summary of the distinctive characteristics of Alpha's application context: integrating constituent lower-level, centralized, real-time subsystems into one system focused on performance of a single real-time mission; and managing that system to meet (in many cases, changing) mission objectives given the current (in many cases, changing) internal and external circumstances. Real-time distributed system integration and mission management is a predominately asynchronous endeavor in which conflicts and overloads are inevitable, but most activities have hard and soft real-time constraints. These combined factors constitute the requirement for an apparent oxymoron: distributed resource management which is dynamic and non-deterministic yet nonetheless real-time.

To help resolve this conflict between needing functional and temporal dependability, and accommodating inherent uncertainty, we devised a new paradigm for real-time computing; it is founded on two concepts. The first is that real-time computations have individual and collective "benefit" (both positive and negative) to the system which are functions of their completion times; thus, maximizing *accrued* benefit can be the basis for highly cost-effective real-time acceptability criteria. The second is that in many (especially mission-level, distributed) real-time computing systems, it may be much preferable for the OS to do the best (as defined by the user) that it can under the current resource and application conditions, than for the OS to fail because these conditions violate the re-

strictive premises underlying its structure and resource management algorithms.

The paper then describes Alpha's kernel programming model, which is based on *distributed threads* that span physical nodes, carrying their real-time and other attributes to facilitate system-wide resource management. Transactional techniques are employed to maintain trans-node application-specific execution correctness and data consistency. We also synopsize the intended system configurations—where Alpha is either the only OS in the system, or supports distributed applications while interoperating with extant centralized OS's and applications (e.g., UNIX or low-level sampled-data subsystems).

Some of our architectural experiences to date with Alpha are then synopsized in the context of comparisons with related kernel work, such as Mach 3.0.

The paper concludes with a brief overview of the project history and status.

2. Distributed Computing And Its Implications On Real-Time Resource Management

Physically distributed computing arises whenever a computing system comprised of a multiplicity of processing nodes has a ratio of nodal computing performance to internodal communication performance (primarily latency but also bandwidth) which is significantly high as far as the application is concerned.

The nodal/internodal performance ratio and its significance—i.e., the degree of physical distribution—will usually be different for computations at different levels in the system. For example, a given system could have ratios which are: relatively insignificant to an application; highly significant to the "middleware" application framework, such as DCE; insignificant to the nodal operating systems; and highly significant to the internodal communication subsystem. The significance of a given ratio may also differ for levels of abstraction within the computations at a particular system level—e.g., within some system level, there may be: object method invocations, to which the ratio is relatively insignificant; built on layered remote procedure calls, to which the ratio is rather significant; which in turn are built on (uniform, location-transparent) message passing, to which the ratio is quite insignificant. The significance of the nodal/internodal performance ratio to a computation—i.e., its degree of physical distribution—depends on intrinsic characteristics of the computation, essentially related to how autonomous the per-node components are, and on the programming model used to express the computation.

The application pull for physically distributed computing in real-time contexts is both involuntary and voluntary.

The most common involuntary motivation is that application assets (e.g., the telecommunications switching offices, the different processing stages of a manufacturing plant, the ships and aircraft of a battle group) are inherently spacially dispersed, and a real-time performance requirement does not permit the latency of the requisite number of communications which would be needed between those assets and a centralized computing facility.

A prominent voluntary reason for physical dispersal is survivability, in the sense of graceful degradation for continued availability of situation-specific functionality. For example, it may be more cost-effective to distribute—i.e., replicate and partition—a telecommunications operation system, or an air/space defense command system, than it is to attempt to implement a physically centralized one which is infallible or indestructible.

There is also a powerful contemporary technology push for physically distributed computing due to the rapid increases in microprocessor performance and decreases in cost. This too is both voluntary and involuntary—the latter is due to current primary memory subsystems being disproportionately slower than processors, making clustered multicomputers attractive; regular topology ar-

ray style multicomputers also exhibit physically distributed computing properties as well.

Because of their physical dispersal, most distributed real-time computing systems are "loosely coupled" via i/o communication (employing links, buses, rings, switching networks, etc.), without directly shared primary memory. This generally results in variable communication latencies (regardless of how high the bandwidth) which are long with respect to local primary memory access times. The nature, locations, and availability of the applications' physical assets often limit the system's viability if it becomes partitioned (unlike, for example, a network of workstations), so these internode communication paths are frequently redundant and physically separated to reduce the probability of that happening.

A typical non-real-time distributed computing system—fitting the workstation model, such as [1])—is a network of nodes, each having an autonomous user executing unrelated local applications with statically hierarchical two-party (e.g., client/server) inter-relationships, supported by user-explicit resource management which is primarily centralized per node. In contrast, a real-time distributed computer system is mission-oriented—i.e., the entire system is dedicated to accomplishing a specific purpose through the cooperative execution of one or more applications distributed across its nodes. Thus, there is more incentive and likelihood for the nodes to have dynamically peer-structured multiparty inter-relationships at the application and OS layers.[1]

Real-time distributed computing applications are usually at a supervisory level, which means that their two primary functions are generally distributed system-wide resource management, and mission management. The former function is the application-specific portion (at least) of the distributed real-time execution environment, which augments the real-time (centralized or distributed) OS to compose the constituent subsystems into a coherent whole that is cost-effective to program and deploy for the intended mission(s). The latter function then utilizes this virtualized computing system to conduct some particular mission. It is far more probable than in a non-real-time dedicated-function system (e.g., for accounting) that the mission's approaches and even objectives are highly dependent on the current external (application environment) and internal (system resource) situation. Many real-time distributed computing applications are subject to great uncertainties at both the mission and system levels ("the fog of war" [3] is the extreme but most obvious example.

These application characteristics, combined with the laws of physics involved in distribution, results in the predominant portion of the supervisory level computing system's run-time behavior being unavoidably asynchronous, dynamic, and non-deterministic.[2] Therefore, even though most of the application results have (hard and soft) real-time constraints, it is not always possible for all of them to be optimally satisfied, nor to exactly know in advance which ones will be [4].

Nonetheless, real-time distributed computing applications and systems are usually *mission-critical*, meaning that the degree of mission success is strongly correlated with the extent to which the overall system can achieve the maximal dependability—regarding real-time effectiveness, survivability, and safety—possible given the resources that are available (in the general sense—e.g., operational, suitable, uncommitted, or affordable). The dependability of lower-level subsystems may be either necessary for mission-critical functions (e.g., digital avionics flight control keeping the aircraft aloft), or part of the uncertainty to be tolerated at the system and mission levels (e.g., communications, weapons); but it is not sufficient (e.g., a flying aircraft which cannot perform its mission is wasting resources and creating risks).

1. *Logically* distributed computing can be defined in terms of the kinds and degrees of multilateral resource management [2].
2. Non-determinism includes stochastic activity as a subset (e.g., Petri nets are the former but not the latter); some non-determinism in distributed real-time computing systems may have, or be usefully considered to have, a probability measure—this can permit more tractable resource management algorithms.

This implies the need for *best-effort* real-time resource management—accommodating dynamic and non-deterministic resource dependencies, concurrency, overloads, and complex (e.g., partial, bursty) faults/errors/failures, in a robust, adaptable way so as to undertake that as many as possible of the most important results are as correct in both the time and value domains as possible under the current mission and resource conditions [5][6][7]. It also entails offering the application user opportunities to at least participate in, if not control, the requisite resource management negotiations and compromises by adjusting his mission objectives and expectations to fit the circumstances, or changing the circumstances (constraints, resources), if either alternative is possible.

The option of best-effort resource management makes possible a choice between very firm á priori assurances of exact behavior in a limited number of highly specific resource and mission situations (as offered by static, highly predictable real-time technology), versus weaker assurances of probable behavior over a much wider range of circumstances. Examples of applications which seem to call naturally for either highly predictable or best-effort resource management come immediately to mind, as do others where the decision is more obviously a value judgement regarding risk and cost management under the exigencies of the situation.[3].

Virtually all such real-time reconciliation of uncertainty and dependability at the system and mission levels has historically depended solely on the talent and expertise of the system's human operators—e.g., in the control rooms of factories and plants, in aircraft cockpits. Increasingly, the complexity and pace of the systems' missions, and the number, complexity, and distribution of their resources, cause cognitive overload which requires that these operators receive more support in this respect from the computing system itself. Application software cannot solely bear this responsibility because the effectiveness of any resource management policy—especially real-time ones—depends on how consistently it is applied to all resources down to the lowest layer hardware and software. Moreover, best-effort policies place special demands on almost all the OS facilities.

The role of traditional real-time computers and OS's has been limited to being automatons in low-level sampled-data subsystems, where this contention between accommodating uncertainty and ensuring dependability does not arise. There, the premise is that the application and system's behavior is (or can be made) highly predictable, allowing extensive á priori knowledge about load and communication timing, exceptions, dependencies, and conflicts. Standard real-time theory and practice is to attempt to exploit such information with static techniques which aspire to provide guarantees about application and system behavior (not just the ends to be achieved, but even the exact means by which they are achieved)—but only under a small number of rigidly constrained, and often unrealistic, mission and resource conditions which are anticipated and accommodated in advance [9]. The classic real-time static, deterministic mindset and methodology constitute a simple special case, usually adequate for its intended domain, which does not scale up to distributed real-time systems.[4]

Therefore, one essential aspect of the research underlying the Alpha kernel was an improved understanding of "real-time" resource management.

3. It is instructive and enlightening to consider this issue in light of the many conclusive demonstrations by cognitive scientists of the ubiquitous human trait to miscalculate risks: for example, because we tend to be probability-blind near the extremes, we judge the annihilation of a risk very differently from the "mere" reduction of that risk, even when the latter diminishes the risk by a far greater degree [8].

4. There are analogous paradigm shifts in nature for larger, more complex systems—e.g.: in physics, where Newton's view of gravity as a force was generalized by Einstein as space/time curvature [10]; and in biology, where higher animals are more complex because they are larger, rather than conversely [11].

3. Real-Time in Alpha

The classical "hard/soft real-time" dichotomy has proven to be unnecessarily confusing and limiting, even for the centralized context in which it arose. We created the Benefit Accrual Model [12][13] to overcome the limitations of the classical one, and especially to facilitate the expansion of real-time computing into distributed systems.This model generalizes Jensen's notion of time-value function resource scheduling [14] (high performance architectural support for them was also initially explored [15]). Time-value function and best-effort scheduling has long been a research topic of the Archons project [16][17][18], and subsequently is being studied by others (e.g., [19][20]); its first OS implementation was in Alpha [17], and it has also appeared elsewhere, such as in Mach [21].

We regard a computation to be a *real-time* one if and only if it has a prescribed completion time constraint representing its urgency—i.e., time criticality—which is one of its acceptability criteria. Therefore, an OS is real-time to the degree that it explicitly (whether statically or dynamically) manages resources with the objective of application (and consequently its own) computations meeting their time constraint acceptability criteria. Thus, according to our definition of a real-time system, physical time, whether absolute or relative, is part of the system's logic—analogous to faults being states in a fault-tolerant system.

A computing system may meet its time constraint criteria without explicitly managing its resources to do so—by being endowed with excess resources (e.g., MS-DOS on a Cray Y-MP is "real fast" rather than real-time), or by good fortune—in which cases the system may fairly be considered to *operate in real-time* (and is not of interest to us).

In the classical "soft" real-time perspective, computation completion time constraints are usually not explicitly employed for scheduling; and in the corresponding "hard" real-time view, activity completion time constraints are defined as deadlines. In our Benefit Accrual Model, time constraints are both explicit and richer to delineate and encompass the continuum from "soft" to deadlines. They are represented with two primary components: the expression of the *benefit* to the system that the results yield, individually and collectively, as a function of their completion times; and application-specific predicates for acceptability optimization criteria based on *accruing* benefit from the results—see Figure 1.

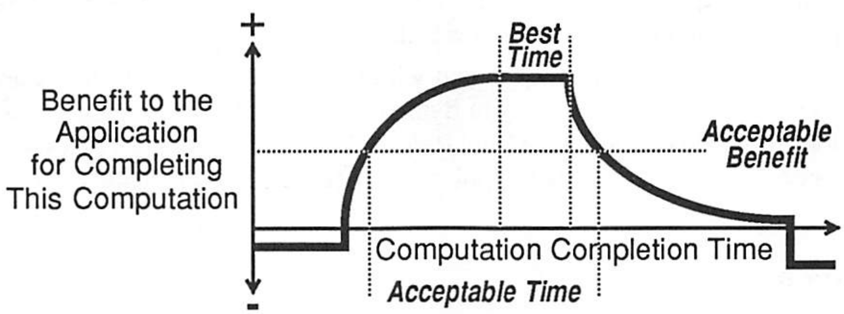


Figure 1: Benefit Function

Alpha's principal real-time strategy is to schedule all resources—both physical (such as processor cycles, secondary storage, communication paths) and logical (such as virtual memory, local synchronizers, and transactions)—according to real-time constraints using the Benefit Accrual Model described in the preceeding subsection.

A uniform approach to resource scheduling allows each α-thread itself to control all the resources it utilizes anywhere in the system—e.g., across nodes, and from user through to devices (such as

performing disk, network, and sensor/actuator accesses). The resulting continuity of the α-thread's time and importance (among other) attributes, together with appropriate scheduling algorithms, ensures coherent maintenance of real-time behavior.

Alpha separates resource scheduling into application-specific policies—e.g., defining optimality criteria and overload management—and general mechanisms for carrying out these policies. The mechanisms, together with a policy module interface, are part of the kernel. There are no restrictions on the kind, or number of, scheduling policies; obviously the parameters, such as time constraints and importance, must be interpreted consistently over a domain of execution, but there may be multiple such domains. The policies may use time directly, as with deadline algorithms, or indirectly, as with periodic-based fixed priority algorithms (e.g., rate-monotonic), or not at all, as with round robin.

For Alpha's context (notably characterized by aperiodicity and overloads in a distributed system), we conceived a new class of *best-effort* real-time scheduling policies [4]. Such policies are non-stochastically non-deterministic according to our taxonomy of scheduling [22].

Best-effort scheduling policies utilize more application-supplied information than is usual, and place specific requirements on the kind of scheduling mechanisms that must be provided. Obviously, resource scheduling which employs more application-supplied information, such as benefit functions, exacts a higher price than when little such information (e.g., static priority) or no information (e.g., round robin) is used. That price must be affordable with respect to the correctness and performance gained in comparison with simpler, less expensive, scheduling techniques.

The effectiveness and cost of a representative best-effort benefit accrual algorithm have been studied by simulation [6][7] and measurement [23][24]. The results demonstrate that this kind of scheduling is capable of successfully accruing greater value than the widely used algorithms—e.g., round robin and shortest-processing-time-first (both non-real-time algorithms), static priority (the most common real-time algorithm), and closest-deadline-first—for loads characteristic of Alpha's intended environment (at least). The scheduling cost per thread and per scheduling decision depend on the specific algorithm and on the implementation of time-value function representation and evaluation. The conclusion of these initial studies was that it is feasible to design and implement best-effort benefit accrual algorithms which provide a greater return to the application for resource investment than if some of those resources were available to the application itself because of lower-cost scheduling. Further experiments, together with research on analytical characterization of the performance of best-effort and benefit function scheduling, are taking place.

If desired, a large part of the price can be paid with the cheap currency of hardware: in multiprocessor nodes, a processor can be statically or dynamically assigned to evaluating the time-value functions (as is done in Alpha Release 1 [25] and Release 2 [26], respectively); or a special-purpose hardware accelerator, analogous to a floating point co-processor, could be employed.

4. Distribution in Alpha

Alpha is a distributed kernel which provides for coherent distributed programming of not only application software but also of the OS itself. It exports a new programming model which appears to be well suited for writing real-time distributed programs. Consequently, it also provides mechanisms having the objective of supporting a full range of client layer trans-node resource management policies; these policies are clients of the kernel and so are not discussed here.

Alpha provides a new kernel programming model because extant ones (cf. [27]) were deemed inappropriate for Alpha's objectives in various ways. For example, message passing and (direct read/write) distributed shared (virtual) memory (e.g., [28]) were rejected as being too low level (i.e., unstructured) for cost-effectively writing real-time distributed programs; distributed shared memo-

ry also suffers from implementation difficulties (e.g., the cost of synchronizing dirty pages). The conventional layered remote procedure call model is client/server oriented and thus imposes disadvantageous server-centric concurrency control; contemporary implementations also have insufficient transparency of physical distribution. Distributed data structures, such as Linda's Tuple Space [29], suffer from access sequencing design decisions—such as non-determinism and fairness—which render them unsuitable for time-constraint ordered tasks. Language-dependent models, like Argus [30], potentially offer significant advantages but are not acceptable in Alpha's overall user community. Virtually all of the few real-time distributed OS programming models are intended only for strictly deterministic systems (e.g., [31][32]).

Alpha's native programming model is provided at the kernel interface so the OS itself, as well as the applications, can employ real-time distributed programming. The OS layer can augment the kernel-supplied programming model in application-specific ways, or substitute an alternative one (e.g., POSIX, although full UNIX compatibility has never been an Alpha goal and therefore may be inefficient).

Alpha's kernel presents its clients with a coherent computer system which is composed in a reliable, network-transparent fashion of an indeterminate number of physical nodes. Its principle abstractions are objects, operation invocations, and distributed threads; these are augmented by others, particularly for exceptions and concurrency control [33][34].

4.1 Objects

In Alpha, objects are passive abstract data types (code plus data) in which there may be any number of concurrently executing activities (Alpha's *distributed threads*); semaphore and lock primitives are provided for the construction of whatever local synchronization is desired. Each instance of an Alpha client level object has a private address space to enforce encapsulation; the resulting safety improvement is judged to generally be worth the higher operation invocation cost in Alpha's application environments (but if performance dictates, objects may be placed in the kernel, as discussed in the Invocation subsection below).

An instance of an object exists entirely on a single node. Alpha's kernel supports dynamic object migration among nodes. Kernel mechanisms allow objects to be transparently replicated on different nodes, and accessed and updated according to application-specific policies.

Alpha objects are intended to normally be of moderate number and size—e.g., 100 to 10,000 lines of code—as dictated by the implemented cost of object operation invocation. The kernel defines a suite of standard operations that are inherited by all objects, and these can be overloaded.

Objects and their operations are identified by system-protected capabilities which provide a network location-independent space of unique names. Capabilities can be passed as invocation parameters.

An object may be declared permanent, which causes a non-volatile representation of its state to be placed in a local crash-resistent secondary storage subsystem, the mechanisms of which are normally (but not necessarily) resident in the kernel. These mechanisms also support application-specific atomic transaction-controlled updates to an object's permanent representation, which are performed in real-time—i.e., scheduled according to the real-time constraints of the corresponding distributed threads. This necessitates that Alpha take an integrated approach to managing resources in accordance with both the time-related, and the particular logical dependency, constraints which define execution correctness and data consistency; most other OS's (whether or not they are real-time and whether or not they are distributed) deal with these two kinds of constraints separately, if at all. Permanent objects obviate the need for a traditional file system in many applications, but any desired file system organization and semantics can readily be provided by client (system or application) layer policies.

In the interests of generality, Alpha's kernel views the universe of objects to be flat; any structure is added by higher software layers.

4.2 Operation Invocations

The invocation of an operation (method) on an object is the vehicle for all interactions in the system, including OS calls. Distributed threads (see the next subsection) are end-to-end computations (not processes or threads confined to an address space) which extend from object to object via invocations. Thus, operation invocation has synchronous request/reply semantics (similar to RPC); operations are block structured.

It is straight-forward to augment (or subvert the intent of) Alpha's synchronous programming model by constructing alternative asynchronous computational semantics on the native mechanisms—e.g., message sends which don't wait, and calls which spawn a concurrent activity that might not return. Asynchronous IPC (like assembly language programming) has a long history of use and staunch supporters in real-time computing. But the same effects can be accomplished in a better behaved manner by proper use of Alpha's model. Its kernel-level invocations are deliberately synchronous (and its threads distributed) because employing asynchrony is generally <u>not</u> straight-forward, particularly for handling all the kinds of concurrency and exception cases which happen in a distributed real-time system. Even non-real-time, centralized multiprocessing OS's whose native IPC mechanisms are asynchronous often seek to improve the intellectual manageability of client programming by also providing layered synchronous facilities. Mach provides examples of this: MIG [35] is used not just for RPC but also sometimes for local IPC, and is the only IPC facility provided in a fault tolerant system built on Mach [36]; an approach to transparent recovery which does use Mach's asynchronous messages is significantly complicated by them [37]. Similarly, the asynchronous message passing communication hardware of large multicomputers is often abstracted into a more productive synchronous programming model with software development tools [38]. Asynchronous RPC was removed from Amoeba 2.0 as having been "a truly dreadful decision" and "impossible to program correctly" [39]. Asynchronous IPC is also highly problematic for attaining the TCSEC B3 level of assurance for multilevel security in OS's (e.g., in Trusted Mach [40]).

Invocation parameters are passed into the invoked object's domain on invocation, and when the invocation is complete, return parameters are passed back to the invoking object's domain. All invocation (request and reply) parameters, except capabilities, are passed by value on the current frame and stack for this invocation by the distributed thread; each distributed thread has its own stack and cannot access the stack of another. Handling bulk data (e.g., [41]) does not seem to be a typical requirement in system integration and mission management applications (a programmer-transparent implementation enhancement facilitates movement by value of large parameters within a node); however, asynchronous bulk data movement can be performed as a kernel client layer service without changing the programming model. We consider procedural parameters contrary to the spirit of object oriented systems. Alpha does not presently deal with the topic of parameter representation conversion which arises among heterogeneous nodes; that problem receives wide attention elsewhere (unlike most of those we are currently focusing on), and since Alpha does not require an especially unique solution, we will adapt one when necessary.

Invocation, not simply message passing, is a fundamental kernel facility of Alpha. Consequently, objects may be placed within the kernel address space for performance improvements. Of course, if they seek further speedup by directly accessing kernel data structures, that forecloses the (sometimes desirable) option of moving them back out of the kernel into client space.

We think of each inter-node invocation as creating a *segment* of that distributed thread. Invocation masks the effects of thread segmentation with unusually strong semantics for independence and transparency of physical distribution [42].

Alpha's kernel performs implicit binding at the time of each invocation (utilizing a protocol, rather than a centralized name server such as the NCS Location Broker [43], for locating the target object). The kernel also includes provisions to optionally perform explicit binding; this is for optimizing performance in relatively static cases (e.g., due to specially located resources), and for other purposes such as testing and failure recovery.

Communication errors are handled by message protocols which may be realized as kernel or client objects. The various motivations for reliable messages being entirely client level functionality (e.g., the microkernel arguments and the end-to-end argument [44]) must be balanced against the acceptability constraints of the particular real-time system. Alpha's communication subsystem incorporates an approach to a general framework and mechanisms for implementing communication protocols, due to the requirement for problem-specificity imposed by real-time requirements [45]; future versions may substitute corresponding concepts and techniques from the *x*-Kernel [46].

Alpha provides orphan detection (presently under the usual assumption of fail-stop nodes) and elimination, at any time (even on a distributed thread which is already undergoing orphan elimination), and in a decentralized manner [45]. The technique employed requires active tracking of the progress of each distributed thread by the Alpha instance on the node where that thread is rooted. However, any orphaned activity can be successfully detected and eliminated, without requiring significantly more complex mechanisms such as transactions or distributed clocks [47]. This technique also allows dynamic trade-offs of communication bandwidth and processing against orphan detection latency. The standard Alpha configuration is for orphan detection and elimination to be kernel functionality, although it can alternatively be implemented in client space if desired.

Invocations may fail for various reasons, such as protection violation, bad parameters, node failure, machine exception, time constraint expiration, or transaction abort. The failure semantics of invocation instances in a real-time distributed system must be application-specific, so Alpha's kernel includes additional mechanisms for defining them; at-most-once is the default. See the subsection below on exceptions.

4.3 Distributed Threads

An Alpha *distributed thread* (α-*thread*) [17] is the locus of control point movement among objects via operation invocation, as shown in Figure 2. It is a distributed computation which transparently and reliably spans physical nodes, contrary to how conventional threads (conceived as lightweight processes) are confined to a single address space in most other recent OS's such as Mach [48] and Chorus [49]; however, Clouds [50] employs a thread model similar to Alpha's.

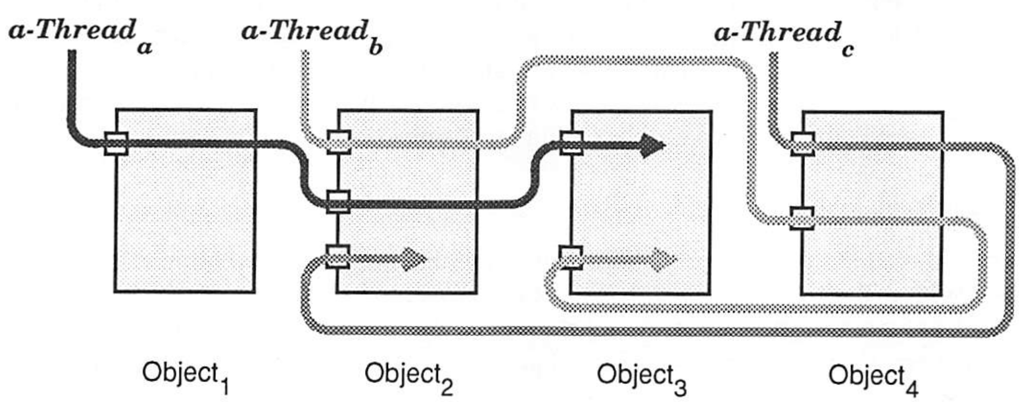


Figure 2: Alpha's *Distributed Threads* (α-*Threads*)

An α-thread carries parameters and other attributes related to the nature, state, and service re-

quirements of the computation it represents. An α-thread's attributes may be modified and accumulated in a nested fashion as it executes operations within objects (illustrated in Figure 3). Unlike how

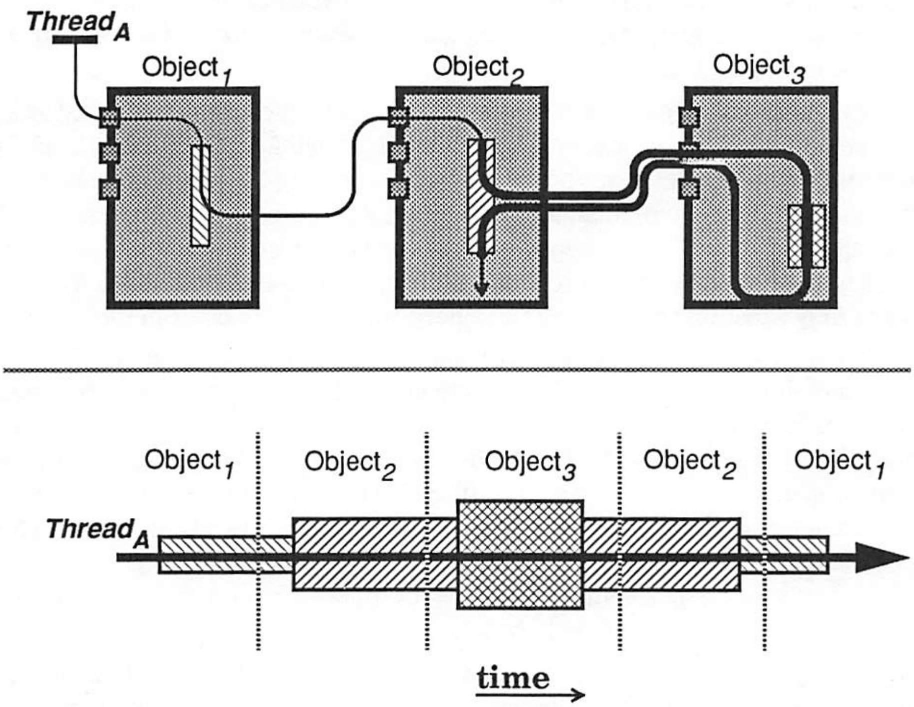


Figure 3: α-thread Attribute Accumulation

RPC or message passing are employed in other OS's, these attributes are utilized by Alpha's kernel and its clients as a basis for performing real-time, system-wide, decentralized resource management.

α-threads are the unit of schedulability, and are fully pre-emptable, even those executing within the kernel. Thus, when the scheduling subsystem detects that there is a ready α-thread whose execution is more likely to increase the accrued benefit than the one currently running, the executing α-thread can be pre-empted by the waiting one. The pre-emption costs and expected completion time of the lower benefit-accrual α-thread are taken into account when making this decision. In addition, Alpha offers scheduling algorithms which explicitly deal with the various kinds of resource dependencies and conflicts among α-threads, and if appropriate, they roll forward or roll back a lower benefit-accrual α-thread which is blocking a higher one [7]. The fully pre-emptable and multithreaded design of Alpha's kernel facilitates real-time behavior and allows symmetric multiprocessing within the kernel itself as well as within its clients.

4.4 Exceptions

Every α-thread is subject to exceptions—an event that interrupts the α-thread's normal execution flow. With respect to an α-thread's execution, an exception may be synchronous (e.g., a machine check) or asynchronous (e.g., a real-time constraint expiration, transaction abort, α-thread break). The kernel's exception handling mechanisms treat synchronous and asynchronous exceptions uniformly.

Alpha's kernel provides exception handling mechanisms defined in terms of kernel-provided abstractions; these language-independent mechanisms can be used by the OS and language run-time systems to construct appropriate exception handling policies, which clients may, in turn, use to es-

tablish application-specific exception handlers (which, for example, retry, perform compensatory actions, or utilize the results attained prior to occurrence of the exception). The mechanisms permit applications to define handlers for the core set of exception types defined by the kernel, and also to define their own exception types and handlers for them.

The mechanism for specifying exception handlers is the *exception block*, a block-structured construct that complements the block-oriented nature of invocations. The `BEGIN` operation for exception blocks opens a scope of execution during which its parameters define the exception handlers to be used for the specified exception types while the α-thread is executing within that block. The `END` operation closes the inner-most open exception block. Like other α-thread attributes, exception blocks may be nested and exception block scoping is dynamic. The exception handling attributes are protected by the kernel, so that subsequent application errors cannot corrupt them.

When an exception of a particular type occurs, control of the α-thread is moved to the handler specified by the inner-most exception block that defines a handler for exceptions of that type. Because Alpha's kernel is fully pre-emptable, an exception may force an α-thread out of the kernel, at an arbitrary point (even if it is blocked), to perform exception handling. So, in addition to any user-defined exception blocks, the kernel treats each operation defined on an object as an implicit exception block. The kernel-defined handlers for these implicit blocks perform only the simple clean-up operations necessary to ensure that the kernel will retain a minimum degree of internal consistency (i.e., it will neither leak resources, nor fail due to inconsistent internal data structures). The existence of this implicit block also ensures that exception blocks opened in one object will not be closed in another (i.e., exception blocks must nest correctly within an object).

An α-thread always handles its own exceptions, preserving the correspondence between the α-thread and the computation it is performing. Following the occurrence of an exception, the kernel adjusts the attributes of the α-thread so that each exception handler is executed with attributes appropriate for the α-thread exception block at that point—among other things, this ensures that the proper scheduling parameters are associated with the exception handling.

The occurrence of a single exception may require multiple levels of exception handling to be performed. An example is a real-time constraint expiration exception, which is not discharged until the exception block level at which the real-time constraint was established is reached. Another example is the elimination of an orphan α-thread segment, where the exception is not discharged until the segment is eliminated. In such cases, exception handlers are executed in order from inner-most to outer-most until the exception is discharged.

If exception processing spans multiple invocations, all invocation frames of the α-thread except the head will be waiting for an invocation to complete. System-level interface libraries can take advantage of this fact to simplify application-level exception processing in these cases.

4.5 Transactions

Transactions are useful for a wide variety of integrity purposes, including the optional extension, when needed, of invocation semantics to zero/one (e.g., [51]). Of particular interest is that Alpha promotes (but is not limited to) a transactional approach to trans-node concurrency control[5] so that collectively α-threads behave "correctly," as defined by the application, and so that distributed (both replicated and partitioned) data remains mutually "consistent," as defined by the application [52]. The many advantages of this include permitting remote invocations to pass mutable parameters by value (which thus constitute shared state), while avoiding the limitations of conventional

5. In distributed systems, synchronization is generally achieved through maintaining an ordering of events, rather than through mutual exclusion as in centralized systems. We do not consider that sending messages to a centralized synchronization entity is consonant with the objectives of distribution.

server-centric concurrency control in network-style distributed systems. Transactions from the database context cannot be simply transplanted into an OS—this is particularly true for real-time systems because they are integrated into the physical nature of the application.

One general problem is that traditional transactions bundle the properties of atomicity, permanence, and serializability together at one (high) performance price. Instead, Alpha's kernel provides transaction mechanisms for atomicity, permanence, and application-specific concurrency control individually; these can be selected and combined at higher (OS and application) layers according to a wide range of different transaction policies whose cost is proportional to their power.

In real-time systems, permanence is not universally desirable: some transactions update data that is relevant only to the local node for this incarnation; and in many cases, the physical world maintains the true state (as related to the system by the sensor subsystems) that is only cached or approximated by the data manipulated by transactions.

Conventional transactions define correctness as serializability, which limits concurrency and thus performance [53]. Alpha's mechanistic technique encourages the use of application-specific information in non-serializable transactions. This allows optimized correctness through customized commit and abort handling: transactions can commit and allow other transactions to observe their results with no ill effect for an arbitrary period of time; their abort processing can also be deferred for an arbitrary period of time (unless there are other mitigating circumstances). Traditional recovery techniques such as rollback and redo, and those requiring the client applications to be deterministic or idempotent (e.g., stateless) [54][55], are not always germane in real-time contexts. Furthermore, performance can be improved through cooperation among non-serializable transactions [56].

The second major limitation of conventional transactions is that they do not have and use information about application result real-time constraints. They are scheduled according to different criteria (e.g., serializability) than are the tasks (α-threads in Alpha's case); they employ locking mechanisms (e.g., time stamps) unrelated to task (α-thread) scheduling; and the time required to acquire and release resources, as well as the time required to commit and abort transactions, is potentially unbounded. To overcome these limitations, Alpha's transactions are real-time, most importantly meaning that they are scheduled according to same application real-time constraints and policies as are all other resources.

4.6 Alpha System Architectures

A distributed OS could impose or accommodate a variety of possible OS configurations and thus system architectures [4]; Alpha is primarily intended for three of these.

The purest form of a distributed OS is for it to be the only OS in the system—native on all nodes—in which case, it must provide local OS functionality as well, and be cost-effective for both local (centralized) and distributed applications. The nodes, and their interconnection, must have sufficient resources to support both local and distributed computations. It is difficult for such an OS to accommodate backward compatibility with extant local OS or application software, but it is the cleanest and most coherent approach when there is the freedom to create an entire system from scratch (as is often the case in real-time applications, especially distributed ones).

The second system architecture of interest to us is for the distributed OS to be native on its own interconnected hardware nodes, forming a *global OS* (GOS) subsystem This necessitates that it provide full local OS functionality as well, although not necessarily in a manner which is most cost-effective for low-level, sampled-data real-time applications. The GOS subsystem nodes physically interface with the local nodes and OS's—which constitute the low-level real-time subsystems—via the GOS subsystem interconnect and/or a system level interconnection. This case offers: superior performance due to local/global hardware (node and interconnect) concurrency; compatibility with heterogeneous and pre-existing local subsystems (OS's and applications); major logistical benefits

from the relative independence of local and global OS and application changes.

The third alternative system architecture for Alpha is for there to be distributed and local OS's which are separate but co-resident on the local nodal hardware. On uniprocessor nodes, co-residency would require something like a virtual machine monitor to create an illusion of two or more processors, which entails overhead (that may be affordable because of the performance of contemporary microprocessors). On multiprocessor nodes, co-residency can be relatively easy and highly effective—Alpha can (and often does) co-exist and interoperate with UNIX on any or all of the system's multiprocessor nodes (as shown in Figure 4), thus making available to Alpha applications

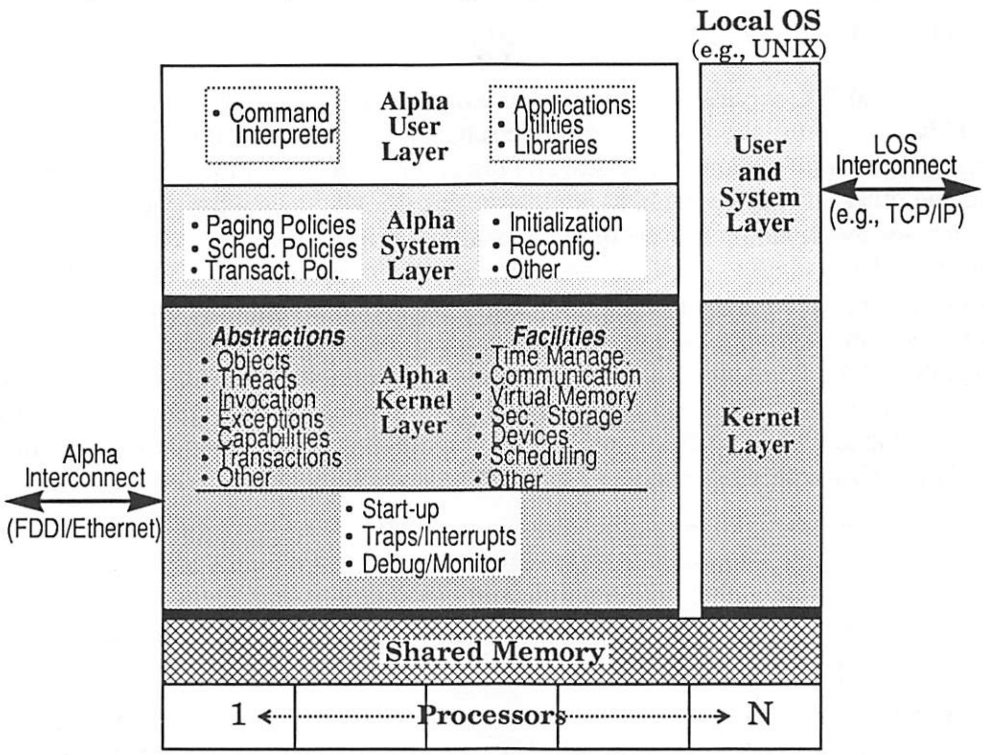


Figure 4: Alpha Co-Resident With UNIX On Multiprocessor Nodes

all the non- (or less) real-time functionality of UNIX (such as GUI's, ISO protocols, and software development facilities) [57]. With either configuration, the internode connection must be able to support both distributed and local OS's, or there must be separate interconnection structures for each type.

5. Multilevel Security in Alpha

The construction of multilevel secure (B3 and higher) [58], distributed, real-time systems is of great interest to a large part of Alpha's prospective user community, and thus is an area of active research within the Alpha project [59]. There are many inherent conflicts between the requirements of real-time and B3 security, including (but not limited to): covert timing channels arising from the real-time scheduling algorithms; covert storage channels due to resource sharing and contention; the potential for malicious denial of service by untrusted applications improperly asserting great urgency and importance; and predictability of security enforcement behavior. These conflicts are likely to require appropriately authorized, situation-specific, dynamic trade-offs between various secu-

rity and real-time characteristics.

6. Architectural Lessons Learned From Alpha

The following synopsizes some of the important lessons we believe that we have learned from our experiences with Alpha's architecture in comparison with that of others, such as Mach 3.0.

6.1 Kernel Support for Distributed Threads

It is common to find RPC services implemented as a layer on top of asynchronous message passing facilities. This layering usually involves multiple scheduling events, complex RPC stubs for argument marshalling, multiple context changes, and the consequent loss of the client's identity, time-constraint, and other attributes. Mach, as well as some other OS's, attempts to overcome some of these deficiencies by providing subsets of the message passing service that are optimized for RPC [60]. In the case of Mach, only simple messages are optimized, but messages with capabilities (port rights) or out-of-line data do not benefit from the optimizations. These optimizations reduce the number of scheduling interactions and system calls necessary to implement RPC, but identity, scheduling information (e.g., priority) and other attributes are not propagated. In contrast, when an Alpha distributed thread moves from object to object, its time constraints and other properties remain in effect. Alpha's kernel is fully pre-emptable, and every effort is made to run the most important ready thread whether it is executing in client space or kernel space.

Timely service interruption processing is essential to Alpha's strategy for scheduling overload situations. Alpha's exception model explicitly takes into account orphans and distributed exceptions. The need for this functionality is not unique to real-time systems; UNIX and POSIX compatible systems also must support interruptible system calls. Orphan detection and elimination is typically not provided by layered RPC facilities.

These limitations of layered RPC facilities make building a distributed real-time RPC facility problematic and inefficient. Recently published work suggests that high performance RPC is best obtained with RPC-specific kernel assistance [60][61][62][63].

Multi-server operating systems have many of the characteristics of distributed applications even if all the servers reside on a single node. The client process communicates with the OS server(s) via IPC. In a standard implementation of UNIX, when an application invokes a system service the client thread of control moves from the user application context into the operating system. When the request for service is completed the thread returns to user space. A variety of attributes follow the thread from user space to the kernel including identity, working directory, quotas, etc. The kernel uses these attributes to track and manage resource consumption, provide interruptible system calls and insure security. The interaction of user applications with standard operating systems is very reminiscent of distributed threads. Mimicking the semantics of "legacy" operating systems, UNIX in particular, with a collection of servers is complicated by microkernels that do not provide sufficient support for distributed programming.

Auditing and authentication forwarding are significant problems for secure distributed systems. Changing identity or subjects during a request for service makes it difficult to associate the server actions or resources with the client responsible for the request for service. This association is important for both authentication and auditing. Distributed threads facilitate this aspect of security by preserving the identity of the distributed computation.

6.2 Dynamic, Adaptive Thread Management

Most kernels, such as Mach and Chorus, provide a threads abstraction that associates each thread with a single task. By default, Mach thread management is static. If servers are over-subscribed, then requests block, regardless of whether there are computation resources available. If the server is under-subscribed, then kernel resources such as process control blocks and other kernel data structures are reserved but under-utilized.Dynamic or adaptive thread management is the responsibility the application designer. Experience has shown that application level solutions to thread management for distributed systems tend to be complex, inefficient and prone to error.

Thread management based on global (inter-task) resource usage and requirements is difficult and not possible without compromising security. Alpha threads do not prevent applications from controlling the number of extant threads, however the default behavior is to create threads dynamically, on an as-needed basis. The kernel, with its knowledge of available resources, has the primary responsibility for balancing the computation demands against the available resources.

It has been argued that thread management and RPC layers can be constructed in user space that provide most of the advantages of distributed threads; this may be true, but the examples that the authors are familiar with are not convincing.

6.3 Protected Capabilities

Alpha capabilities are kernel protected and have context sensitive names, similar to Mach port rights and port names. Other systems, such as Chorus and Amoeba, provide unprotected capabilities. While the cost of invoking either type of capability seems roughly equal, our experience confirms the assertion that unprotected capabilities are somewhat less expensive to pass as parameters. However, unprotected capabilities are insufficient to build high trust systems [64][65]. Though early versions of Alpha associated capabilities with objects (similar to the way Mach associates ports and port rights with tasks), we found this awkward and inconvenient. Subsequent versions of Alpha permit capabilities to be associated with threads. Thread local capabilities simplify capability management. They enable the construction of secure subsystems and can be leveraged in other ways when building secure real-time systems [59].

6.4 Object Invocation Via Broadcast

Alpha uses broadcast protocols aggressively to locate and invoke remote objects. Each node maintains a list of objects that are local to the node—the object dictionary. When an object invocation is broadcast, each node receives the message and examines its dictionary to determine if the object being invoked resides locally. This results in simple object location protocols with a relatively constant time (a useful property for real-time applications). While the time required to locate the object is small, the broadcast processing overhead imposed on nodes can be significant. If the dictionary of objects is too large to fit in memory then it must be paged. Paging would add significantly to the total overhead and the variance of broadcasting object invocations.

Other OS's, and Mach in particular, have demonstrated that if the kernel "owns" capabilities, it is possible to track and cache the current location of any capability or object. Though Alpha currently broadcasts each remote object invocation, we have done a preliminary investigation into caching remote object location information as one means to reduce the number of object invocation generated broadcasts.

6.5 Separation of Policies from Mechanisms

The separation of policies from mechanisms is more than code words for "layered design." The design of mechanisms and policies is an approach to encapsulation and layering that results in rela-

tively simple mechanisms suitable for the implementation of a variety of policies.

Our experience suggests that the kernel interface is not the only interesting mechanism/policy boundary. We have found that the creation of policy modules within the kernel—such as for scheduling, secondary storage, and communications—was of great value. The scheduling subsystem is not simply layered; a common set of mechanisms has been used to implement a number of significantly different real-time and non-real-time scheduling policies. The encapsulation of these policies facilitated not only their development and maintenance but also their wholesale replacement.

7. Project History and Status

Alpha arose as the first systems effort of Jensen's Archons Project on new paradigms for real-time decentralized computer systems, which began in 1979 at Carnegie-Mellon University's Computer Science Department. Design of the Alpha OS itself was started in 1985 and the initial prototype ("Release 1") was operational at CMU in the Fall of 1987.

Alpha is a native kernel (i.e., on the bare hardware), which necessitates a great deal of effort on low level resource management. But much of that work involves fundamental issues in Alpha's design and implementation—the usual research approach of emulating an OS at the application level (typically on UNIX) would have introduced excessive real-time distortions due to the vast disparity between the programming model and structure of Alpha's kernel and those of UNIX. Alpha's first hardware base was multiprocessor nodes built from modified Sun Microsystems and other Multibus boards; the nodes were interconnected with Ethernet.

The principle goal of the Archons Project in general and its Alpha component in particular was both to create new concepts and techniques for real-time distributed OS's, and to validate those results through industrial as well as academic experimentation. The first step in that validation process was to augment our own personal experience with industrial real-time distributed computing systems by involving a user corporation early in the development of the initial Alpha prototype. Because Archons and Alpha were sponsored primarily by the DOD, and because the leading edge real-time computing problems and solutions always arise first in defense applications, we looked to the DOD contractor community for our initial industrial user partnership. We selected General Dynamics' Ft. Worth Division, exchanging application and OS technology in the highest bandwidth way—by exchanging people. Their C^3I group successfully wrote and demonstrated a real-time distributed air defense application on Alpha in 1987 [23], and their avionics organization intended to base the mission management OS of a planned new aircraft on Alpha's technology.

Once the proof of concept prototype was operational, we sought to begin transition of Alpha's technology into practice by establishing a relationship with a computer manufacturer. To further facilitate that transition, the leadership and then the staff of the Alpha project moved to industry in 1988. The intent is for Alpha to serve as a technology development vehicle—first for application-specific real-time distributed operating systems (e.g., for telecommunications, simulators and trainers, C^3I, combat systems) where extensive functionality (such as fault tolerance) and high real-time performance are of the utmost importance, no off-the-shelf products exist, and no standards are foreseeable for a number of years. Subsequently, the technology will be available for migration into other OS contexts. A second generation Alpha prototype design and implementation was delivered to several government and industry laboratories for experimental use; the first of these was installed in June 1990. The current version is initially available on MIPS R3000-based multiprocessor nodes interconnected by Ethernet; ports to other hardware are planned. Alpha is non-proprietary.

Alpha research is ongoing at CMU, and related research and technology development is also being conducted cooperatively with several other academic and industrial institutions. The Alpha project is also engaged in partnerships with a number of U.S. and European corporations and other

organizations to develop experimental Alpha applications in the areas of telecommunications and defense systems.

8. References

[1] Anderson, D.P., D. Ferrari, P.V. Rangan, and S.-Y. Tzou, The DASH Project: Issues in the Design of Very Large Distributed Systems, report no. UCB/CSD 87/338, U. of CA/Berkeley EECS Department, January 1987.

[2] Jensen, E.D., *Decentralized Control*, Distributed Systems: An Advanced Course, Springer-Verlag, 1981.

[3] Clausewitz, C. von, On War, tr. by J.J. Graham, N. Trubner & Co. (London), 1873.

[4] Jensen, E.D., *Alpha: A Real-Time Decentralized Operating System for Mission-Oriented System Integration and Operation*, Proceedings of the Symposium on Computing Environments for Large, Complex Systems, University of Houston Research Institute for Computer and Information Sciences, November 1988.

[5] Jensen, E.D., C.D. Locke, and H. Tokuda, *A Time-Value Driven Scheduling Model for Real-Time Operating Systems*, Proceedings of the Symposium on Real-Time Systems, IEEE, November 1985.

[6] C.D. Locke, Best-Effort Decision Making for Real-Time Scheduling, Ph.D. Thesis, CMU-CS-86-134, Department of Computer Science, Carnegie Mellon University, 1986.

[7] Clark, R.K., Scheduling Dependent Real-Time Activities, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, 1990.

[8] Kahneman, D., P. Slovic, and A. Tversky (Ed.), Judgement Under Uncertainty: Heuristics and Biases, Cambridge University Press, 1982.

[9] Stankovic, J.A., Tutorial on Hard Real-Time Systems, IEEE, 1988.

[10] Einstein, A., Relativity: The Special and the General Theory, Crown, December 1916.

[11] Haldane, J.B.S., *On Being the Right Size*, Possible Worlds and Other Essays, Chatto and Windus, 1927.

[12] Jensen, E.D., *A Benefit Accrual Model of Real-Time*, Proceedings of the 10th IFAC Workshop on Distributed Computer Control Systems, September 1991.

[13] Jensen, E. D., *A Benefit Accrual Model of Real-Time*, submitted for publication, 1991.

[14] Stewart, B., Distributed Data Processing Technology, Interim Report, Honeywell Systems and Research Center, March 1977.

[15] Gouda, M.G., Y.H. Han, E.D. Jensen, W.D. Johnson, and R.Y. Kain, *Towards a Methodology for Distributed Computer System Design*, Proceedings of the Texas Conference on Computer Systems, IEEE, November 1977.

[16] Jensen, E.D., *The Archons Project: An Overview*, Proceedings of the International Symposium on Synchronization, Control, and Communication, Academic Press, 1983.

[17] Northcutt, J. D., Mechanisms for Reliable Distributed Real-Time Operating Systems—The Alpha Kernel, Academic Press, 1987.

[18] Jensen, E.D., Alpha—A Non-Proprietary Operating System for Mission-Critical Real-Time Distributed Systems, Technical Report TR-89121, Concurrent Computer Corp., December 1989.

[19] Chen, K., *A Study on the Timeliness Property in Real-Time Systems*, Real-Time Systems, September 1991.

[20] Chen, K. and P. Muhlethaler, *Two Classes of Effective Heuristics for Time-Value Function Based Scheduling*, Proceedings of the 12th Real-Time System Symposium, IEEE, 1991.

[21] Tokuda, H, J.W. Wendorf, and H.Y. Wang, *Implementation of a Time-Driven Scheduler for Real-Time Operating Systems*, Proceedings of the Real-Time Systems Symposium, IEEE, December 1987.

[22] Jensen, E. D., *A Taxonomy of Real-Time and Distributed Scheduling*, Course Notes, Distributed Real-Time Systems: Principles, Solutions, Standards, Paris, June 1991.

[23] Maynard, D.P., S.E. Shipman, R.K. Clark, J.D. Northcutt, R.B. Kegley, B.A. Zimmerman, and P.J. Keleher, An Example Real-Time Command, Control, and Battle Management Application for Alpha, Technical Report TR 88121, Archons Project, Computer Science Department, Carnegie-Mellon University, December 1988.

[24] Northcutt, J.D., R.K. Clark, D.P. Maynard, and J.E. Trull, Decentralized Real-Time Scheduling, Final Technical Report, Contract F33602-88-D-0027, School of Computer Science, Carnegie-Mellon University, February 1990.

[25] Northcutt, J. D., R.K. Clark, S.E. Shipman, and D.C. Lindsay, The Alpha Operating System: System/Subsystem Specification, Archons Project Technical Report #88051, Department of Computer Science, Carnegie-Mellon University, May 1988.

[26] Reynolds, F.D., J.G. Hanko, and E.D. Jensen, Alpha Release 2 Preliminary System/Subsystem Description, Technical Report #88122, Concurrent Computer Corporation, December 1988.

[27] Tannenbaum, A.S. and R. van Renesse, *Distributed Operating Systems*, Computing Surveys, ACM, December 1985.

[28] Cheriton, D.R., H.A. Goosen, and P.D. Boyle, *Paradigm: A Highly Scalable Shared-Memory Multicomputer Architecture*, Computer, IEEE, February 1991.

[29] Kaashoek, M.F., H.E. Bal, and A.S. Tannenbaum, *Experience With The Distributed Data Structure Paradigm in Linda*, Proceedings of the Workshop on Distributed and Multiprocessor Systems, USENIX Association, October 1989.

[30] Liskov, B. and R. Scheifler, *Guardians and Actions: Linguistic Support for Robust, Distributed Programs*, Transactions on Programming Languages and Systems, ACM, July 1983.

[31] Gudmundsson, O., D. Mosse, K-T. Ko, A.K. Agwrawala, and S.K. Tripathi, *MARUTI: A Platform for Hard Real-Time Applications*, Proceedings of the 1989 Workshop on Operating Systems for Mission-Critical Computing, IOS Press, 1992.

[32] Kopetz, H, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, *Distributed Fault-Tolerant Real-Time Systems: The MARS Approach*, Micro, IEEE, February 1989.

[33] Northcutt, J.D. and R.K. Clark, The Alpha Operating System: Programming Model, Archons Project Technical Report TR-88021, Carnegie-Mellon University, February 1988.

[34] Northcutt, J.D., R.K. Clark, S.E. Shipman, D.P. Maynard, E. D. Jensen, F.D. Reynolds, and B. Dasarathy, *Threads: A Programming Construct for Reliable Real-Time Distributed Programming*, Proceedings of the International Conference on Parallel and Distributed Computing and Systems, International Society for Mini- and Micro-Computers, October 1990.

[35] Draves, R.P., M.B. Jones, and M.R. Thompson, MIG: The Mach Interface Generator, Internal Working Document, Computer Science Department, Carnegie-Mellon University, November 1989.

[36] Chen, R. and T.P. Ng, *Building a Fault-Tolerant System Based on Mach*, Proceedings of the USENIX Mach Workshop, October 1990.

[37] Goldberg, A., A. Gopal, K. Li, R. Strom, and D. Bacon, *Transparent Recovery of Mach Applications*, Proceedings of the USENIX Mach Workshop, October 1990.

[38] Wu, M.-Y. and D.D. Gajski, *Hypertool: A Programming Aid for Message-Passing Systems*, Transactions on Parallel and Distributed Systems, IEEE, July 1990.

[39] Tannenbaum, A.S., R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, J. Jansen, and Guido van Rossum, *Experiences With the Amoeba Distributed Operating System*, Communications of the ACM, December 1990.

[40] Branstad, M.A., H. Tajalli, F. Mayer, and D. Dalva, *Access Mediation in a Message-Passing Kernel*, Proceedings of the IEEE Symposium on Security and Privacy, May 1989.

[41] Gifford, D.K. and N. Glasser, *Remote Pipes and Procedures for Efficient Distributed Communication*, Transactions on Computer Systems, ACM, August 1988.

[42] Levy, E. and A. Silberschatz, *Distributed File Systems: Concepts and Examples*, Computing Surveys, ACM, December 1990.

[43] Apollo Computer Corp., Network Computing System Reference Manual, 1987.

[44] Saltzer, J.H., D.P. Reed, and D.D. Clark, *End-to-End Arguments in System Design*, Transactions on Computing Systems, ACM, November 1984.

[45] Northcutt, J.D., and R.K. Clark, The Alpha Operating System: Kernel Internals, Archons Project Technical Report #88051, Department of Computer Science, Carnegie Mellon University, May 1988.

[46] Hutchinson, N.C., and L.L. Peterson, *The x-Kernel: An Architecture for Implementing Network Protocols*, Transactions on Software Engineering, IEEE, January 1991.

[47] Herlihy, M.P., and M.S. McKendry, *Timestamp-Based Orphan Elimination*, Transactions on Software Engineering, IEEE, July 1989.

[48] Rashid, R., *Threads of a New System*, Unix Review, August 1986.

[49] Rozier, M., V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herman, C. Kaiser, S. Langois, P. Leonard, and W. Neuhauser, CHORUS Distributed Operating Systems, CS/TR-88-7.8, Chorus Systemes, 1989.

[50] Dasgupta, P., R.C. Chen, S. Menon, M.P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R.J. Leblanc, W.F. Appelbe, J.M. Berabeu-Auban, P.W. Hutto, M.Y.A. Khalidi, and C.J. Wilkenloh, *The Design and Implementation of the Clouds Distributed Operating System*, Computing Systems, USENIX, June 1988.

[51] Brown, M.R., K.M. Kolling, and E.A. Taft, *The Alpine File System*, Transactions on Computer Systems, ACM, November 1985.

[52] Jensen, E.D., *The Implications of Physical Dispersal on Operating Systems*, Proceedings of Informatica '82, Sarajevo, Yugoslavia, March 1982.

[53] Garcia-Molina, H., *Using Semantic Knowledge for Transaction Processing in a Distributed Database*, Transactions on Database Systems, ACM, June 1983.

[54] Borg, A. et al., *Fault Tolerance Under UNIX*, Transactions on Computer Systems, ACM, January 1989.

[55] Ravindran, K., and S.T. Canson, *Failure Transparency in Remote Procedure Calls*, Transactions on Computers, IEEE, August 1989.

[56] Sha, L., E.D. Jensen, R. Rashid, and J.D. Northcutt, *Distributed Co-Operating Processes and Transactions*, Proceedings of the ACM Symposium on Data Communication Protocols and Architectures, Mar. 1983, and Synchronization, Control, and Communication in Distributed Computing Systems, Academic Press, 1983.

[57] Vasilatos, N., *Partitioned Multiprocessors and the Existence of Heterogeneous Operating Systems*, Proceedings of the USENIX Winter 1991 Conference, January 1991.

[58] Gasser, M., Building A Secure Computer System, Van Nostrand Reinhold, 1988.

[59] Loepere, K.P., F.D. Reynolds, E.D. Jensen, and T.F. Lunt, *Security for Real-Time Systems*, Proceedings of the 13th National Computer Security Conference, October 1990.

[60] Draves, R., *A Revised IPC Interface*, Proceedings of the USENIX Mach Workshop, October 1990.

[61] Karger, P.A., Improving Security and Performance for Capability Systems, Technical Report No.149, Computer Laboratory, University of Cambridge, October 1988.

[62] Bershad, B.N., T. E. Anderson, E. D. Lazowska, H. M. Levy, *Lightweight Remote Procedure Call*, Transactions on Computer Systems, ACM, February 1990.

[63] Schroeder, M.D., and M. Burrows, *Performance of Firefly RPC*, Transactions on Computer Systems, ACM, February 1990.

[64] Anderson, M. R. D. Pose, and C. S. Wallace, *A Password-Capability System*, The Computer Journal, February 1986.

[65] Karger, P. A. and A. J. Herbert, *An Augmented Capability Architecture to Support Lattice Security and Traceability of Access*, Proceedings of the 1984 Symposium on Security and Privacy, IEEE, April 1984.

9. Acknowledgments

The research on Alpha and its constituent technology at Concurrent Computer Corp., SRI International, and Carnegie Mellon University is sponsored by the U.S.A.F. Rome Laboratories, Computer Systems Branch. Additional support for Alpha is supplied by Digital Equipment Corp., and was provided at CMU by the U.S. Naval Ocean Systems Center, and the General Dynamics, IBM, and Sun Microsystems corporations.

The views contained in this paper are the authors' and should not be interpreted as representing those of the sponsoring organizations.

The authors are grateful to J. Duane Northcutt for his invaluable leadership of Alpha's first prototype design and development, and to Ed Burke, B. Dasarathy, Jim Hanko, Don Lindsay, Dave Maynard, Martin McKendry, Doug Ray, Sam Shipman, George Surka, Jack Test, Jeff Trull, Nick Vasilatos, Huay-Yong Wang, and Doug Wells for their essential contributions to Alpha. Teresa Lunt and Ira Greenburg at SRI International, together with Ray Clark at Concurrent Computer Corp. and Doug Wells at the Open Software Foundation, are the principals on the B3 multilevel secure version of the Alpha kernel. Alan Downing and Mike Davis of SRI International are the principals on the integrated resource management project for Alpha, assisted by Jon Peha at CMU.

Performance of the BirliX Operating System

P.Schüller, H.Härtig, W.E. Kühnhauser, H. Streich
German National Research Center For Computer Science (GMD)
Schloß Birlinghoven, P.O. Box 1316, D-5205 St. Augustin
email: kuehnhsr@gmdzi.gmd.de

Abstract

This paper is a detailed performance analysis of the BirliX Operating System. BirliX is an operating system for distributed, secure, and fault-tolerant applications. Unlike many other operating system kernels that support lower-level abstractions like threads and memory segments, the BirliX kernel supports abstract data types (adts) as the basic interface paradigm. BirliX servers are instances of adts that live in an controlled environment: all BirliX types share type-independent properties like a common communication paradigm, or security and fault-tolerance mechanisms that are integrated into the kernel.

This paper discusses the implications of the BirliX kernel architecture and its adt paradigm on the kernel performance. Several other implications concerning security and fault tolerance are discussed in [KHKL90, KH90, HKLR92, KHK+ 91].

On top of the BirliX kernel, nine BirliX Types emulate the bsd 4.3 Unix interface. While the main interest lies in the performance analysis of the security and fault tolerance mechanisms within the kernel, we also included performance evaluation of the Unix emulation in the paper to make the system - concerning this conventional domain - comparable to others.

1 BirliX Kernel Concepts

This is a short introduction into the concepts of the BirliX kernel. A more detailed summary can be found in [HKLR92, KHK+91].

In the last decade, research in distributed operating systems has come up with kernel/server - architectures consisting of a small kernel and a server layer. While the kernel provides a framework for the integration of servers, the servers on top of it implement high level operating system services.

Kernel/server architectures have several advantages, such as flexibility, portability and an apt paradigm for distribution. Nevertheless, new architectures also raise new questions. For correctness validation, portability and maintainability, kernels must be small, so the criteria for what has to be in the kernel might be very restrictive. On the other hand, the kernel is the universal framework for the integration of all servers ever, and its mechanisms and server paradigm must both be powerful (to achieve generality) and restrictive (to achieve a uniform system view).

The key kernel interface paradigm in the BirliX system is the *BirliX Type*. The kernel is basically a BirliX Type management system with mechanisms for definition and instantiation of types, and the communication of instances. BirliX Types share a common set of type-independent attributes and functionality inherited from a single system-defined *primary type*.
A very natural feature of any type management system is the functional extensibility and adaptation to new requirements by simply defining new types.

Concerning distribution, the BirliX Type paradigm provides the smallest unit of identification and communication, the *Type Instance*. Application programs on BirliX are sets of interacting instances distributed among several nodes in a computer network. Instances are identified and located by name servers on top of the kernel; name servers map user-level names to system-level names and provide valuable locating informations. Instances communicate via calling each others operations.

The underlying communication mechanism is a network-transparent rcp that for efficiency, security and fault tolerance reasons maintains a client/agent relationship between caller and callee.
At the application level, a network of BirliX hosts appears as a single large system providing computing resources without regard to the multiplicity and distribution of the machines which actually provide the resources.

Fault tolerance is based on checkpointing and recovery of Type instances. The Primary Type exports type-independent checkpointing and recovery mechanisms that are small and fast, based on copy-on-write memory sharing. These mechanisms may be tailored to type-specific needs by each individual BirliX Type.
Based on these mechanisms, server-layer policies like replication, transactions and migration are implemented.

BirliX security is based on the encapsulation of instances, access control lists, subject restrictions, and authentic messages. Encapsulation gives access to instances only via well-defined operations. Access control lists defend individual instance operations against calling subjects, subject restrictions limit the world visible to individual subjects, and authentic messages guarantee unforgeable caller ids for checking rights in access control lists.

1.1 BirliX Types

In many ways, BirliX Types are similar to Eden Types [ABLN85] or Smalltalk classes.
BirliX Types have type descriptions that contain a type interface (signature) and methods (code of exported operations).
BirliX Type instances communicate by calling each others operations. There is no other way for accessing an instance.
Each instance has a unique identifier; mapping of high-level user names to unique identifier is done by name servers that themselves are Type instances. Unique identifiers are independent of instance attributes as type or location. Location information is provided by name servers as hints. The establishment of a client/agent communication connection (binding) is done by a type-independent *TypeManager* within the Primary Type.
Instances can be active on their own; they may contain *threads* that run in parallel to each other within the same instance.
Instances are persistent. They continue to live as long as they are referenced, e.g. by a name server or a binding from another instance. If neither a binding nor internal activities exist, an instance releases processor and main memory resources; a passive representation is created and written to permanent storage. Any new binding to a passive instance reactivates the instance. Activation and passivation is done by the *TypeManager*.
Instances are not isolated from system crashes, but they provide checkpoint and recover primitives for creating checkpoints (additional passive representations) and reactivating instances from checkpoints. Passive representations can be sent to other locations. Thus, checkpoint and recover primitives are also used as the basic primitives for type-independent instance migration.
Protection of instances is based on access control lists, subject restrictions, and authentic messages. Access control lists protect individual instance operations vs. calling subjects, subject restrictions limit the world visible to individual subjects, and authentic messages guarantee unforgeable caller ids for checking rights in access control lists.

1.2 Kernel Interface

Like many systems with kernel/server architecture, the BirliX kernel provides two basic hardware abstractions for storage (segments) and processor (threads) resources, and a paradigm for assembling low-level resources to higher-level "objects" (Amoeba [TM81], Chorus [Cho89], Mach 3.0 [Ras86]). Unlike many systems with kernel/server architecture, the BirliX kernel paradigm are abstract data types with rigorous encapsulation and type-checking rules.

BirliX Types

BirliX Types are implemented by a type-independent general implementation structure called a *team*. A team is a collection of threads sharing an address space, and a collection of segments mapped into that address space. A team provides storage and computing resources needed by an instance in its active phase.

When a passive instance becomes active, a team is created using the information in the passive representation. When an active instance becomes passive, a new passive representation is written to disk, and the team is destroyed.

Depending on their role within an instance, threads are called *agents* or *natives*. Communication bindings are maintained in *access descriptors*.

Agents

As result of a communication request, a communication binding is established between the calling and the called instance. A communication binding is maintained in an *access descriptor* within the calling instance and an agent within the called instance. The agent authenticates its client, controls its access permissions and performs the client's operations on the instance.

Access descriptors contain the UId of the called instance, locating information to rebind to the agent after crashes, and an agent address (host and thread id) that is used directly by the underlying message passing system. Agent addresses may become invalid due to host crashes or instance migration.

The one-to-one relationship of access descriptors and their agents is fundamental for transparent recovery of communication bindings after migration or system crashes.

Natives

Natives are internal activities of an instance (e.g. a local garbage collector) that run asynchronously to agents and other natives.

Figure 1 shows two instances with their agents, natives, segments and access descriptors. A single binding exists.

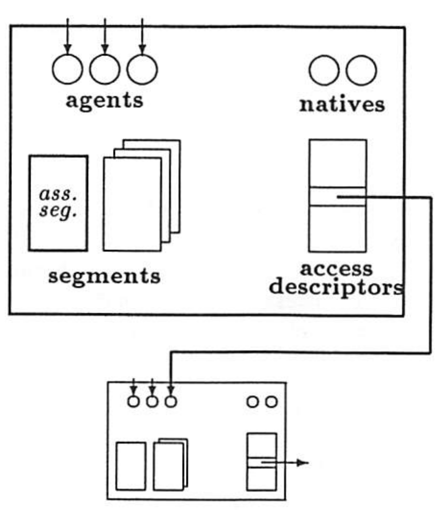


Figure 1: Teams

Segments

Segments represent all memory-like physical resources. A small number of different segment types exist. *Ordinary segments* provide persistent storage via pages of permanent storage and use main memory page frames as cache. *Associated segments* are ordinary segments that contain the data representation of passive instances. To enforce instance autonomy, associated segments are access-able only by the Type Manager. *Physical segments* are allow direct access to physical memory such as the frame memory of a high resolution display.

1.3 Security

While the inadequacy of the security measures of many environments is agreed upon [Hog88], there still is a lack of well understood security policies in a non military environment. To deal with that situation BirliX is designed in such a way that integrating new security policies is relatively easy.

The basic kernel features for supporting BirliX Types are the *encapsulation of instances* and the *authenticity* of rpcs. For fine-grained access control to instances on a per-operation level, *access control lists (ACLs)* and *subject restrictions* exist.

Access control lists

An access control list defines the access rights of a using instance (*subject*) to a used instance (*object*).

Instances are tagged by a 3-tupel consisting of their unique id, the unique id of their type description (the BirliX Type) and the unique id of the *human user* who created the instance. Whenever an instance is created, the new instance inherits its user tag from the creating instance.

Each ACL entry is a unique id with corresponding access rights; the unique id may identify a (human) user, a type or an instance. Access rights can be granted (positive access rights) or excluded explicitly (negative access rights).

If a user is contained in an ACL the entry defines the access rights of all instances having the responsible user as part of the tag. If a type is contained in an ACL the entry defines the access rights of all instances of the type. If an ACL contains a user or a type, the protection domain may become too large. In that case access rights can be excluded explicitly. Access rights are computed by building the union of positive access rights and subtracting all negative access rights.

In the UNIX emulation the access rights of types in ACLs are used to dispose of the `setuid` mechanism.

Subject Restrictions

Even in the presence of well administrated ACLs the protection domain may become too large. If a new subject that is considered to be suspect is introduced into a system, it may be impossible, to supplement all ACLs with the necessary negative access right. In that case subject restrictions can be used. Subject restrictions restrict the access rights of a suspect instance or all instances of a suspect type. A subject restriction is a tag together with a list of instances that are visible to the tagged instance.

Both, types and instances can be suspect. If an instance is marked suspect, all instances created by the subject inherit the subject restrictions. If a type is marked suspect, all instances of that type are restricted by the type's subject restrictions.

The primary type implements subject restrictions by suppressing calls to instances that are not covered by the list.

Message Authenticy

Since any ACL-based protection policy requires safe knowledge of the callers id, authenticity of messages (rpcs) is maintained with respect to to a given access descriptor/agent relationship. To build up an access descriptor/agent relationship the subject's instance identification and its label are passed encrypted to the newly created agent. Subsequently each message contains a capability securely identifying the sender. The capability is built encrypting a checksum and sequence number using a single key encryption scheme. Keys are exchanged between stations using a public key encryption scheme.

1.4 Recovery

In many newer research systems, recovery-based fault tolerance is supported by providing a transaction management facility (sometimes called atomic action). The transaction management is either part of the kernel [L+87] or implemented on top of the kernel supported by kernel primitives [PW85].

Application-specific fault tolerance policies differ in the way the consistency of the whole application (a recovery line) is achieved. Transaction oriented recovery policies leave the definition to the programmer by allowing him to denote the beginning and the end of the critical region of the program. The policy then guarantees the *at most once* property. Other policies may consider arbitrary points of execution, e.g. triggered by a chosen time interval, as a recovery line for all participating instances. That especially makes sense in long time transactions where intermediate states are kept for performance reasons. Another area of application for that policy is long number crunching application in highly parallel systems, where the probability of fault for a single node is high and where it cannot be afforded to stop a complete computation for the fault of one node. In that case the policy cannot rely on the cooperation of programmers.

The BirliX kernel provides mechanisms for checkpointing and recovery of Type instances. The mechanisms are small and type-independent, based on copy-on-write sharing of memory pages. For performance enhancement, the mechanisms may be tailored to type-specific needs by each individual BirliX Type. Based on the mechanisms, high-level fault tolerance policies can be implemented on the BirliX Type layer.

In general, the type-independent checkpoint operation creates a passive team representation by collecting all *relevant state* in the associated segment and afterwards checkpointing the segment. Checkpointing an associated segment is a kernel primitive using copy-on-write sharing, both for main memory pages and mass storage.

1.5 The Unix Emulation

On top of the BirliX kernel the Unix bsd4.3 interface is emulated by a set of 9 BirliX Types [1]. Besides these types, additional name server types provide a network-wide name space (implementing the Andrew File Server naming scheme) and the integration of *Unix Process Types* in the common name space. Types for user administration provide a network-wide unique mapping of user names to system-level unique ids and allow individual users to define their personal authentication method together with mandatory authentication tools (e.g. chipcards).

Unix process types run a native with two coroutines, one executing the user-level program code (defined by a Unix a.out format) and one emulating the Unix system calls. A system call by the user-code coroutine transfers control to the emulation coroutine that maps a system call to rpcs to BirliX Type instances. E.g. a Unix *read* system call on a file is transformed into a call to the involved Unix File Type instance. The *fork* system call creates a new *Unix process type* instance and initializes it with the contents of the caller's associated segment (copy-on-write sharing). The *exec* system call replaces the user coroutine.

[1] the *Unix Process Type*, the *Unix File Type*, the *Unix Directory Type*, the *Unix Pipe Type*, the *Unix Socket Type*, and the device types *Mouse Type*, *FrameBuffer Type*, *CharTerminal Type*, and *BlackHole Type*

Several advantages from the adt approach are also available via the Unix emulation. As Unix processes and files are implemented as BirliX types, both can be checkpointed, recovered and are subject to fine-grained security mechanisms. On the other hand, generality has its price. In the current BirliX implementation, the very special Unix operation to create a new process (*fork*) is implemented by the same kernel primitives that are used for checkpointing. Forking or checkpointing a Unix process takes about 50 milliseconds on a SUN 3/60, which is about twice the time SUN OS needs for a fork.

1.6 Related Work

BirliX is in several ways related to other operating systems. Abstract data types are also found in Eden [ABLN85]. While Eden is implemented on top of the UNIX operating system, BirliX is a persistent object management system with a standard operating system interface (4.3BSD UNIX) emulated by an application program on top of it.

Kernel/server architectures are also found in many other operating systems; some examples are [TM81, Cho89, Ras86].

The overall design of BirliX was influenced by the EUMEL system [Lie87], its recovery primitives were influenced by experience with the PROFEMO [NKK86] transaction manager.

Parts of this summary are a taken from [HKLR92].

2 Performance Evaluation

The performance evaluation of the BirliX kernel focusses on operations on BirliX Types and on parts of the Unix bsd4.3 emulation. Especially we will focus our analysis on the operations to create, activate, checkpoint, and deactivate instances, and on the basic security mechanisms (primary operations). For these operations, the major activity points within the kernel will be identified.

We will identify three sources of influence on the system performance: on is due to the strictly object-oriented approach, one to the chosen kernel structure, and the third is due to too simple implementations.
Examples for the performance of the Unix emulation are taken from the emulation of the Unix file system and the process management.

2.1 Environment

Practical work was done on a Sun 3/60 with 12 MB of main-memory (8 MB for system usage, 4 MB for storing measurement results) and a Micropolis SCSI-Harddisksystem. High resolution timing up to 1 μs is provided by the AM9513A timer-chip. The Sun-Interface of that chip was developed in Berkley by P. Danzig and S. Melvin [DM90]. For data collection, an event-driven monitor has been used, which is part of the BirliX kernel. The monitor operates in the event-driven-batch mode [Jai91]. Events are described by 4-tupels, consisting of a time stamp (1 μ resolution), the identification of the executing thread, the identification of the event and an optional parameter. Events are generated by injecting monitor calls into the kernel source code.

To keep interference low, events are written into a reserved main memory area. All high-level chaching mechanisms (name caches, caches for open objects) were disabled.

2.2 Primary Operations

The efficiency of a BirliX Type strongly depends on its implementation level. The type description of *Kernel Level Types* are part of the kernel code; all instances of kernel types share a common

address space. Communication and instanciation of kernel types is cheap, but a high level of trust in the implementation is needed. Typical examples for kernel-level types are file servers or standard name servers.

User Level Types have user-defined type descriptions that are dynamically created and that are maintained in type-description objects. Instances of user level types usually are protected by a separate address spaces. A typical example of a user level type from our Unix is any Unix program: an executing Unix program is emulated by a BirliX instance running in e separate (user level) address space.

Instance Activation

When a passive instance becomes active, a team is created using the information in the passive representation. When an active instance becomes passive, a new passive representation is written to disk, and the team is destroyed.

Figure 2 shows in detail the operations within the kernel for the activation of user level type instances.

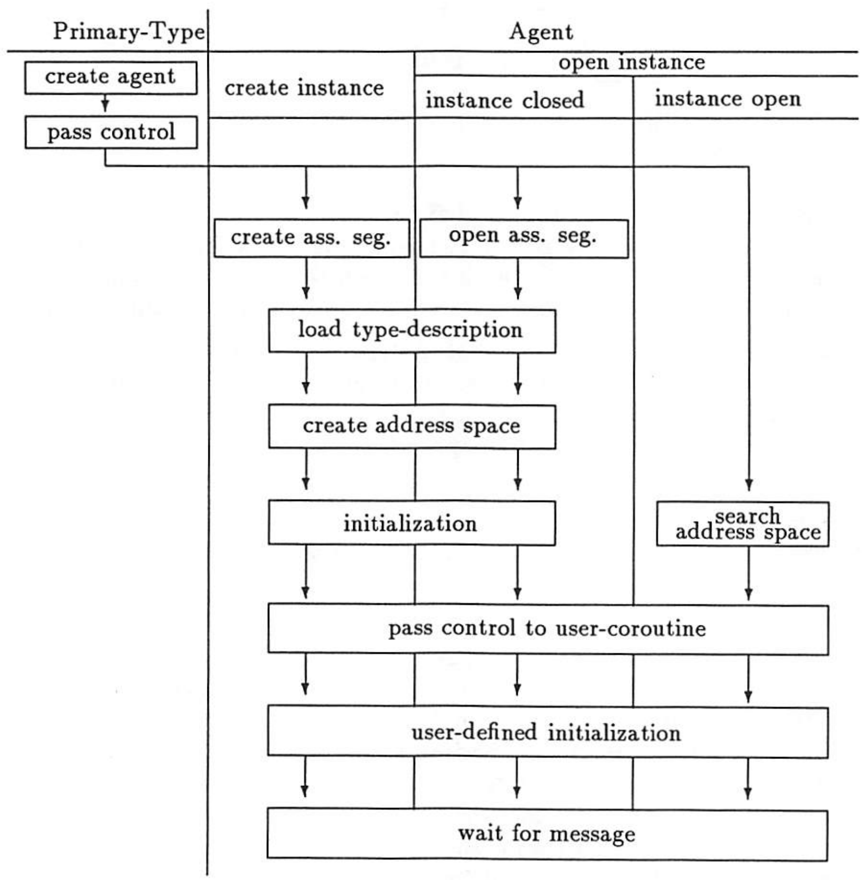


Figure 2: Instance-Activation

Activation of kernel level type instances is more simple. The loading of the type description and the creation and initialization of the user address space is obsolete, because the type-description is part of the kernel code and instances of kernel level types use the kernel address space as their workspace. The sharing of the kernel address space makes communication between kernel types much faster, because procedure calls instead of rpcs are used.

Table 1 shows the detailed measurements of a kernel level type activation. To avoid the influence of type-dependent code an empty kernel level type was used that consisted only of the inherited primary operations.

Operation	*Create*	*Open*	
		Instance closed	*Instance open*
allocate/search Agent-Descriptor	192	299	186
allocate/search Team-Descriptor	348	669	289
init Team-Descriptor	225	278	-
create IState-Monitor	404		
create ass. Segment	1365	-	-
open ass. Segment	-	1418	-
create acl cache	617	652	647
init ACL	125	-	-
init SRL	141	-	-
user-defined init	186	391	49
other	707	811	621
total time	4310	4518	1792

Table 1: Activation Time of Kernel Level Types (in μ-seconds)

The table shows that about 15-20% of the total time is needed for the allocation of the team's administration data structures ("*descriptors*"). A finer grain of evalution showed that these costs are caused by the generic implementation of hash tables which use a finer grain of synchronization than needed here. Another 11.9% of the activation time is spent in other table operations.

The second outstanding value is the time for creating a (Hoare-like) monitor ("$IState - Monitor$") that synchronizes concurrent instance activities during checkpoint operations. A finer grain of evaluation showed that about 18% of the total activation time is spent in monitor operations. No experience has yet been gained whether semaphores are here the better-suited synchronization primitive.

The main part of the total time is used to create resp. open the associated segment of the instance. A finer grain of memory management evaluation has shown that again the generic hash table implementation is responsible for these costs.

In Table 2 the measurements for the activation of user level instances are shown (the table abstracts from a few low-level operations). In contrast to kernel level instances, agents are threads in stead of simple descriptors, type-descriptions are loaded on demand, and a user address spaces is created and initialized. The time for loading of the type-description and the initalization of the address space is by far the major part of the total activation time. As a consequence, the total time of 128 ms for creating a user level instance is about 30 times the time needed for a kernel level type. Nevertheless, the Clouds system [DAM+88] needs 93 ms for the first invocation of a type specific operation (on the same hardware). BirliX and Clouds have different object models and can't be compared directly, but it seems to be worth noting that the order of magnitude is the same.

Considering these results, two small and simple optimizations reduce the activation time by 31 ms:

First, duplicating the type descriptor is implemented via sharing by copy-on-write. In general, type descriptions never are modified, and a simple sharing of the write-protected type description is sufficient, reducing the costs by 11 ms.

Second, the table shows that the most expensive part of a user level instance activation is the initialization of the user address space. Here, the handling of 10 allocation pagefaults needs about 43 ms (an allocation pagefault provides a new, zero-filled, 8k page). An insular tuning of the page

Operation	Create	Open	
		Instance closed	Instance open
create Agent + pass control	2778	2731	2775
open Typdescription-Segment	1479	1184	-
duplicate Typdescription-Segment	10867	-	-
create/open Stack-Segment	1167	1230	-
create user address space	112	97	-
validate user address space	129	155	102
create user-part of Agent	618	617	609
pass control + general initialization + user-defined initialization	57756	60934	2807
open new instance	11731	-	-
close new instance	5951	-	-
total time	128866	91288	10082

Table 2: Activation Time of User Level Types (times in μ-seconds)

fault handling mechanism within the memory management system reduced the time for handling an allocation pagefaults from 4.3 ms to 2.3 ms.

Security Mechanisms

Calling an operation of an instance includes inspecting the instance's access control list (see chapter 1.3) to check whether the client has the necessary right. As the size of an access control list may be very large, agents maintain a temporary cache containing a client-specific acl excerpt. The cache is created during the establishment of a communication binding, i.e. when the agent is created.

The time to create the cache is about 650 μs, which is about 20% of the total time to create a kernel level instance and 14% to open a closed kernel level instance with an acl having 12 entries.

The time needed for cache creating grows linear with the number of actual acl entries, which makes the kernel security mechanisms expensive when acls become big. One consequence is that we are now adopting a more basic kernel security mechanism that just glues a security policy to an instance.

Checkpointing And Recovering

The kernel supports basic mechanisms to checkpoint an instance and fall back to a checkpoint. Both mechanisms are small and type-independent, based on copy-on-write sharing of memory pages (see chapter 1.4). For performance enhancement, the mechanisms may be tailored to type-specific needs by each individual BirliX Type. Without type-specific adaptation, the checkpoint and recover - mechanisms are an overkill, as they — as the are type-independent — do too much work: the kernel mechanisms completely suspend an instance before creating a checkpoint and throw away everything before falling back to it.

The time needed to create and recover a checkpoint of a user instance is shown in table 3.

Additionally, for the type-independent kernel mechanisms to work there is a fix amount of overhead glued to every instance operation caused by state checks that amounts to 81 μs. Again, only those instances should be punished that actually use the checkpoint/recover mechanisms. A similar gluing mechanism as discussed for the security mechanisms would eliminate these costs.

Operation	*time in ms*
Create Checkpoint	71-86
Recover Checkpoint	72

Table 3: Faulttolerance-Operations of User-ADTs.

Instance Closing

Table 4 show the costs creating a passive representation. Due to the synchronous releasing of resources, deleting an instance costs as much as creating a passive representation.

Operation	*Close Instance*		
	create passive repr.	*only agent deleted*	*full instance deleted*
System Level Types	2275	1008	2604
User Level Types	31640	5988	33522

Table 4: Closing a Communication Binding (in μ-seconds)

The *open/close* Paradigm

As a general scheme, operations on instances can be called after some client establishes a binding to an agent within the accessed instance. This scheme is based on the (classical) assumption that *if* an instance is accessed at all, it is almost always accessed more than once by that client (e.g. a compiler opens the source code file, reads the file via several read operations, and finally closes it). Assuming that usage pattern, the *open*-operation can be used to collect and cache several connection-oriented information for routing, communication fault detection, etc.

In object-oriented systems this usage pattern becomes less frequent; in many cases objects are accessed only once (getting a file from a file server, delivering a mail to the mailer, a file to a spooler, or making an entry in a calendar). Even in non-object-oriented systems like Unix, a single access to a directory (looking up an entry) is a frequent operation.

As a consequence, operations on instances must be supported by a more light-weight mechanism that can be adapted to the specific needs of a client. If needs be, a client then may glue additional functionality (like communication fault-detecting mechanisms) to the basic instance communication mechanism.

As an example for the overhead caused by the current *open/some operation/close* - scheme table 5 shows the needed operations to create a Unix-File in the current directory.

As the table shows the actual directory must be opened to insert the new file. After the file-instance is created, the instance of the directory must open the file to increment the reference counter and then close the file. At the end the directory-instance is closed again. We see that the time to increment the reference counter of the file is absolutely dominated by the open/close bracket.
If a file is created by a longer pathname than every directory given in the path must be opened and closed to look for the next one. Table 6 shows how the open-close paradigm effects the time to open/read 8k/close a file in the actual directory and in the directory a/b/c in BirliX and in other systems.
To reduce the costs of activating passive representations BirliX uses an *Instance-Cache* that physically keeps instances open when they are logically closed. The Instance-Cache only has effects on instances that were opened before they are opened again, so that this cache is no general solution for reducing the effects of the open-close paradigm.

Operation	time in μs
invocation	440
open actual directory	1904
check write-permission	650
create file-instance	3935
open file-instance (directory)	1730
increment ref-counter	100
close file-instance (directory)	929
close actual directory	733
total	10421

Table 5: Creation of a Unix-File

System	*Open/Read 8k/Close "foo"*	*Open/Read 8k/Close "a/b/c/foo"*	*Factor*
BirliX	7.7	15.8	2.0
SunOS 4.0	2.8	3.2	1.1
MACH 2.5	3	5.5	1.8
MACH 3.0	5.5	6.6	1.2

Table 6: Effects of the Open-Close Scheme (times in ms)

2.3 The Unix Emulation

As examples for the performance of the Unix emulation on top of the BirliX kernel, some results from the emulated file system and the process management a given. Again, all caches (name caches, instance cache) were disabled.

File System Performance

For evaluating the overall file system performance we used the *modified Andrew File System Benchmark* [Ous90] which was originally developed by M. Satyanarayanan [HKM+88] and was later modified by J. Ousterhout. The benchmark stresses directory and file creation, file copy, file search and compilation activity. A detailed description can be found in [HKM+88, Ous90]. Table 7 shows the results of the benchmark for several systems running on Sun 3/60. The values for the MACH X.X systems are taken from [GDFR90].

System	*dir-create*	*file-copy*	*rec. file stats*	*rec. file read*	*compile*	*total*
BirliX	3	35	45	62	336	481
SunOS 4.0	6	17	16	28	328	395
MACH 2.5	4	20	13	26	336	399
MACH 3.0	1	20	24	34	332	411

Table 7: Modified Andrew File System Benchmark. (time in seconds)

As we see BirliX is about 40-60% slower as the other systems as far as file copy, file stats and file read is concerned. The reason for that can be found in the open/close paradigm discussion. The benchmark uses some Unix tools as cp, find, grep and wc. Most of the additional time compared to other systems was caused by pathname transformation when travelling through the directories, which, for each directory on the path, included one open/lookup/close sequence.

Process Management Performance

For evaluating the costs of process management in BirliX we analyse especially the fork and exit system calls. Table 8 shows the values for the fork/exit pair. The *fork* system call creates a new *Unix process type* instance and initializes it with the contents of the caller's associated segment (copy-on-write sharing). The *exec* system call replaces the user coroutine. The table shows that the pure fork is slightly slower than the fork/exit of MACH 3.0. The developers of MACH 3.0 stated that the fork is responsible for the time of 48 ms and the time for exit is uninteresting. In BirliX the exit system call takes about 30 ms for a minimal process size, which is caused by the synchronous deallocation of the team resources. The use of garbage collection methods seems to be the better way to release resources.

System	*System calls (father)*	*time in ms*
BirliX	fork+exit	$48.4 + 29.7 = 78.1$
SunOS	fork+exit	27.6
MACH 2.5	fork+exit	16
MACH 3.0	fork+exit	48
Amoeba	fork	169.5
Sprite	fork	13.6

Table 8: Costs for *fork* and *exit*.

In BirliX the duplications of the segments of the process team are responsible for the overall time for a fork. To keep administration of such short-living segments low, we now use a delayed-allocation mechanism for secondary storage. Table 9 shows the improvements of delayed allocation concerning the fork system call are about 60%.

Process-Size	*System calls Father*	*System calls Child*	*Normal*	*Improved*
0 kb	fork+wait	exit	98	71
256 kb	fork+wait	read every page+exit	271	198
1024 kb	fork+wait	read every page+exit	961	605
1024 kb	read every page+fork+wait	read every page+exit	1800	970

Table 9: Process-Management Costs (in ms).

3 Lessons Learned

This paper described a straight-forward implementation of an object oriented operating system, i.e. an operating system that provides other services on top of a persistent object system. That implementation — and especially an emulation of a Unix interface on top of it — has been thoroughly analysed for its performance and compared to a vanilla implementation of Unix and to current micro kernels.

Although the performance of a local Unix emulation on top of BirliX (sort of worst case scenario) is lower than a monolithic and micro kernel implementations, the approach taken turned out to be a positive experience. The analysis showed that the weaker performance can be attributed to a large extend to poor coding and structuring decisions (class 2 and class 3 weaknesses), that originated in assumptions not valid any more in systems build along the object oriented programming paradigm. Significant performance improvements have been achieved by only small and insular optimizations.

The remaining conceptual problem is to deal properly with the open/close semantics of the object oriented kernel interface.

A significant restructuring effort, based on these results and on kernel structuring experiences as described in L3 [Lie92], shall be undertaken and and is expected to deliver significantly increased performance.

4 Status

BirliX is a running system. Release 91.1 of 30. June 1991 is running on SUN 3 workstations and has been distributed to several German universities. A port to a 386-based system was done in 1990 by Technical University of Darmstadt, ports to Sparc- and MIPS-based machines are currently done at other university sites. Release 92.1 is scheduled for May 1992.

On top of the bsd4.3 Unix emulation, most standard UNIX utilities are available, including a rich set of public domain software from GNU, XWindows and Motif.

The kernel and the Unix emulation are free of any Unix licence. Most standard Unix utilities are public domain software from GNU.

Papers dealing in more detail with persistence, security and BirliX Types are [KHKL90, KH90, LHK89].

References

[ABLN85] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The eden system: A technical review. *Transactions On Software Engineering*, 11(1), January 1985.

[Cho89] Chorus Systems. *Chorus Distributed Operating System*, Feb 1989.

[DAM+88] P. Dasgupta, R. Ananthanarayanan, S. Menon, C. Chen, and A. Mohindra. Distributed Programming with Objects and Threads in the Clouds System. Technical report, Department of Computer Science and Engeneering, Arizona State University, Tempe, AZ 85287-5406, 1988.

[DM90] P.B. Danzig and S. Melvin. High reultion timing with low resolution clocks and a microsecond resolution timer for sun workstations. *Operating System Review*, 24(1):23–26, January 1990.

[GDFR90] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an Application Program. In *Proceedings of Summer Usenix*, June 1990.

[HKLR92] H. Haertig, W.E. Kuehnhauser, W. Lux, and W. Reck. Operating System(s) on Top of Persistent Object Systems – The BirliX Approach. In *Proceedings of the 25th Hawaii International Conference on System Sciences*. ACM, January 1992.

[HKM+88] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West. Scale and Performance in a Distributed File System. In *ACM Transactions on Computer Science 6(1)*. ACM, February 1988.

[Hog84] J.F. Hogg. Aspen System Overview. Amdahl Corporation, Santa Clara, CA, October 1984.

[Hog88] C.B. Hogan. Protection Imperfect: The Security of some Computing Environments. *Operating System Review*, 22(3):7–27, July 1988.

[Jai91] R. Jain. *The Art of Computer System Performance Analysis.* John Wilney & Sons, Inc., New York, 1991.

[KH90] O.C. Kowalski and H. Härtig. Protection in the BirliX Operating System. In *Proceedings 10th International IEEE Conference on Distributed Computing Systems*, Paris, May 1990.

[KHK+91] W.E. Kuehnhauser, H. Haertig, O.C. Kowalski, W. Lux, and H. Streich. The BirliX Operating System Project. In *Proceedings of the 1991 ERCIM Workshop on Distributed Systems*, Lisboa, Portugal., 1991.

[KHKL90] Winfried E. Kühnhauser, H. Härtig, O.C. Kowalski, and W. Lux. Mechanisms for Persistence And Security in BirliX. In John Rosenberg and J.Leslie Keedy, editors, *Proceedings of the International Workshop on Security And Per sistence*, Bremen, May 1990. Springer.

[L+87] B. Liskov et al. *Argus reference manual.* MIT Laboratory for Computer Science, Cambridge, Mass., 1987.

[LHK89] Wolfgang Lux, Hermann Härtig, and Winfried E. Kühnhauser. On the Implementation of Abstract Data Types in BirliX. In *Progress in Distributed Operating Systems and Distributed Systems management*, volume 433 of *Lecture Notes in Computer Science*, pages 87–109, Berlin, 1989. Springer.

[Lie87] J. Liedtke. *EUMEL - A Portable Operating System for Microprocessors Based on the High Level Language ELAN.* Informatik Kolleg. GMD, March 1987.

[Lie92] Jochen Liedtke. Fast Thread Management And Communication Without Continuations. In *Proceedings of 1992 Usenix Workshop on Micro-Kernels And Other Kernel Architectures*, Seattle, Washington, April 1992.

[NKK86] E. Nett, J. Kaiser, and R. Kröger. Implementing a General Error Recovery Mechanism in a Distributed System. In *Proceedings of the 16th International Conference of Fault-Tolerant Computing, Vienna*, pages 124–129, 1986.

[Ous90] J. Ousterhout. Why aren't Operating Systems Getting Faster as Fast as Hardware? In *Proceedings of Summer Usenix*, June 1990.

[PW85] G.J. Popek and B.J. Walker. *The LOCUS Distributed System Architecture.* Computer Systems Series. The MIT Press, 1985.

[Ras86] Richard Rashid. *From RIG to Accent to Mach: The Evolution of a Network Operating System.* Carnegie Mellon University, Pittsburgh, PA, Computer Science Department, May 1986.

[TM81] Andrew S. Tanenbaum and Sape J. Mullender. An Overview of the Amoeba Distributed Operating System. *ACM Operating Systems Review*, 15(3), July 1981.

Multimedia/Realtime Extensions for Mach 3.0

Jun Nakajima, Masatomo Yazaki, and Hitoshi Matsumoto

Distributed Systems Laboratory
Fujitsu Laboratories, LTD.

{nakajima, yazaki, matumoto}@flab.fujitsu.co.jp

Abstract

To support multimedia applications that have realtime constrains, we have extended the Mach 2.5 and Mach 3.0 operating systems. Our extensions consist of asynchronous event notification, preemptive deadline-driven scheduling, user-mode device drivers, and a temporal paging system(TPS). Preemptive deadline-driven scheduling preempts asynchronous event notification for continuous media that requires each event handler to execute as soon as possible and schedules other event handlers to meet their time constraints as well. In comparison to the version of Mach 2.5, the micro-kernel version reduced unexpected delays in the kernel when other tasks execute expensive system calls. However, the micro-kernel was not a perfect solution to reduce all the unexpected delays in the kernel; operations on I/O devices are still done in kernel mode and sometimes delay unexpectedly. This paper introduces our extensions, and, focusing on realtime issues, points out the merits achieved by moving from the "fat kernel" to the micro-kernel environment, remaining problems, and solutions to them.

1 Introduction

In the current distributed computing systems, it is difficult to integrate multimedia applications with various media, including continuous media, in a general purpose environment. The motivation behind our multimedia extensions came from involvement with an attempt to develop multimedia applications on a traditional UNIX environment. One of the serious problems with the developing attempt was to implement event handlers with time constraints. For example, an interactive music system requires a MIDI(musical instrument digital interface) handler to output MIDI commands at specific intervals; if the handler outputs delayed commands for a note or misses them, the music stumbles. The user-level handler computes the commands dynamically. Hence we need a periodic event notification facility to implement such an application. Because of deficiencies in the traditional UNIX signal mechanism, we should use asynchronous event notification for this purpose. The proposed priority scheduling [6], however, lacks predictability, and the system designers must map a set of specified realtime constraints into a priority order in such a manner that all event handlers will meet their deadlines [12]. Another problem was related to realtime access to virtual memory in multimedia applications that have huge data segments, such as images. Since page fault handling causes unpredictable delays, specific regions of the

address space need to be wired to physical memory pages. It is, however, impractical to "pin-down" such huge data segments.

The extensions we present, called Multimedia/Realtime Extensions, solve these problems. The goal of our extensions is to support multimedia applications in a traditional UNIX environment, where other processes might run simultaneously, and to provide a flexible environment to build such applications. Preemptive deadline-driven scheduling preempts asynchronous event notification for continuous media which requires each event handler to execute as soon as possible, and schedules other event handlers to meet their time constraints as well. TPS manages physical memory pages in the two-dimensional address and time space; a same physical memory page can be shared by multiple pages, if they are not used simultaneously.

We implemented our extensions on both Mach 2.5 and Mach 3.0. The reason why we extended both is historical, and comparing the experiences gives lessons on the micro-kernel when implementing soft realtime systems.

2 Multimedia/Realtime Extensions

Multimedia applications are implemented using integrated and coordinated processing by multimedia handlers. System designers and multimedia developers assume that a multimedia application will require the operating system to support event notification (interrupts that are generated by the multimedia devices). In addition, they assume that processes will require the event handlers to perform asynchronous operations in parallel and in a realtime manner, if necessary.

In our extensions, specific user-level functions(event notification functions), in the multimedia handlers are invoked upon notification of an occurrence of certain events. Occurrences constituting an event are: asynchronous I/O completion, timer expiration, user-defined events, and interrupts from hardware devices. Event notification functions are similar to the UNIX signal functions, but each of them is implemented as a thread to meet the following requirements:

- Realtime asynchronous event notification,
- Parallel execution of handlers,
- Realtime scheduling of handlers according to their time constraints and qualitative requirements

Realtime threads are identical to ordinary ones, except that they are scheduled to meet their realtime constraints.

These constraints are determined by the media type as well as the devices. We divided multimedia devices into two types, *deadline-driven* and *event-driven*, according to their realtime constraints. A *deadline-driven device* requires both its operations and the arrival of data to meet a deadline. If operations to the device, including the transference of data, are completed before a deadline, the media never deteriorates. A typical deadline-driven device is a graphic display. An *event-driven device* requires its operations to execute immediately after the occurrence of the event. The device may have a deadline as well. The media may deteriorate as the response time increases. A typical event-driven device is a MIDI device. Any multimedia hardware device whose status is unknown, except for the time elapsed since it started, is considered as an event-driven device.

In a single processor operating system kernel, dynamic realtime scheduling, such as deadline-driven scheduling, provides high processor utilization and versatility [7, 8]. It is, however, difficult for a user task to find an appropriate deadline so that the thread is scheduled as soon as possible and meets the deadline. Preemptive deadline-driven scheduling in our extensions is an improved deadline-driven scheduling algorithm especially for multimedia/realtime areas. One of its characteristics is that it can preempt a thread for an event-driven device, while scheduling the other threads for deadline-driven devices to meet their realtime constraints. Such a preemption may lower the capacity of deadline-driven scheduling, but the effect will not be serious under the assumption that ordinary event-driven devices, such as a MIDI device, have very small or no buffers, and the handlers can be completed within a relatively short period.

Each realtime thread Th_i is characterized by the following parameters in the kernel:

- start time S_i
- deadline D_i by which Th_i must be completed
- estimated worst case execution time C_i
- thread-type flag F_i to indicate whether Th_i is standard or pressing
- weight W_i which determines the relative importance of Th_i among all the threads.

These parameters are given from information that application programs provide by event definition and event option definition structures. The event definition structure is given in Figure 1. The *evt_handler* element is a pointer to an *event notification function.* The

void	(**evt_handler*)();
void	**evt_value*;
evt_class_t	*evt_class*;
evtset_t	*evt_classmask*;
evtopt_t	**evt_option*;

Figure 1: Event definition structure

evt_value element is an application-dependent value to be passed to the application at time of event notification. The *evt_class* specifies the event class for the event. The *evt_classmask* specifies the set of event classes blocked if and when the event notification function executes.

The *evt_option* specifies the realtime constraints for the event notification function. The definition of the *event option definition structure* is given in Figure 2.

timespec_t	*evt_dtime*;
timespec_t	*evt_etime*;
int	*evt_weight*;
evt_type_t	*evt_type*;

Figure 2: Event option definition structure

The *evt_dtime* specifies the time interval between an occurrence of the event and the deadline. The *evt_etime* is an estimate of the worst case execution time. The flag *evt_type* specifies the event type. C_i, F_i and W_i are given by this structure. D_i is calculated by adding *evt_dtime* to the time that the event occurs.

The implementation of asynchronous event notification of an interrupt is similar that of user-level device management on Mach 3.0. When an interrupt occurs, a small interrupt routine does the minimum set of operations on the device in the kernel-mode so that user-level operations can follow them. When returning from the small interrupt, the kernel invokes the preemptive deadline-scheduling by posting the AST(asynchronous system trap).

3 Comparison

In our extensions, asynchronous notifications to event asynchronous notification functions must be done in a realtime manner without interference from other ordinary non-realtime tasks. Such interference, which could unexpected delays in the kernel, is mainly made by the system calls and interrupt handlers that serve the tasks. Since event notification functions run in user-mode, they need to wait for the completion of the kernel-mode services that a specific event interrupts. Mach 3.0 has an advantage over Mach 2.5 when bounding the amount of time spent within kernel interrupt handlers to the negligible range. This is because user-level device management available on Mach 3.0 reduces the amount of code that runs in kernel-mode [4]. In addition, the UNIX services themselves are preemptive in the Mach 3.0 environment.

This section examines improvements achieved in the new environment, comparing the event dispatch latencies on the two kinds of Mach operating systems. An event dispatch latency is the time interval between the occurrence of an event and the execution of the first instruction of the event notification function in response to the event. The first version of Multimedia/Realtime Extensions was implemented on Mach 2.5. Because the Mach 2.5 kernel is not preempted to run another thread while executing code for a system call, asynchronous event notification was sometimes unexpectedly delayed [9]. To solve this problem, we moved to the Mach 3.0 environment.

The measurements were taken in a user-level MIDI handler, which ran in parallel with tasks that could interfere with it by continuously executing the following primitive operations:

- Spin loop — spins in a loop. This can not interfere with the MIDI handler, but gives the minimum event dispatch latency.
- VM operation and trap — allocates a region of user address space, modifies it and deallocates it. Access to the newly allocated memory causes a trap.
- Read/write — reads and writes a large file, to give the same effects of the interrupt handling for disks.
- Fork operation — creates a new task by *fork*(), which is one of the most expensive system calls in a traditional UNIX.
- Ethernet handling — receives Ethernet packets, to give the effects of the interrupt handling for Ethernet. One packet(1KB size) was sent to the machine every 20ms.

The event notification function in the MIDI handler is invoked upon the timer expiration, that occurs every 120ms in these experiments. The system activities, number of traps, SCSI interrupts, and Ethernet interrupts were also observed. Each measurement was taken more than 1,000 times on an FM TOWNS, a personal computer with an i386 processor (about 2 MIPS) and 8 megabytes of memory. Table 1 and the figures attached in the appendix show measurements for 1,000 event notifications.

3.1 Benchmark Results

Our measurements, presented in Table 1, show that most of event dispatch latencies were reduced in the new environment. This is because the UNIX services themselves are preemptive in the Mach 3.0 environment, and because the micro-kernel does not hold interrupt locks for long periods of time.

Table 1: Event dispatch latencies in a MIDI handler and interrupts

Combination(MIDI +)	Min	Max	Avg	trap	SCSI	Ethernet
	Time(μs)			Number		
Mach 2.5						
spin loop	714	11800	888	19283	72	76
VM and trap	880	11798	1750	41525	72	83
read/write	696	14650	785	344	42408	75
fork	706	12940	907	23180	192	51
Ethernet handling	608	11802	745	196	98	5969
Mach 3.0						
spin loop	766	4192	923	19	24	51
VM and trap	774	4998	1321	58118	69	53
read/write	774	10520	1432	8146	10661	41
fork	766	2962	940	28	75	47
Ethernet handling	760	6838	3334	26	39	6234

The measurements also show that asynchronous event notifications were still delayed unexpectedly, especially when other tasks were executing I/O operations. To investigate these delays, we checked out the system activities when maximum delays were made. We found out the following problems:

- The timer expiration interrupt nests in the disk(SCSI) interrupt. The next realtime thread cannot run until the disk interrupt handler completes, which occurs asynchronously and could cause unpredictable delays. Since the time for holding the disk interrupt lock on Mach 3.0 is comparable to that on Mach 2.5, almost no improvement of performance is achieved for this case.

- Interrupts could occur even after the kernel posts the AST to force the preemptive deadline-driven scheduling, because the duration when all the interrupts are blocked should be as short as possible; commands to a certain device consist of such a sequence of operations that their intervals should be short enough to be acknowledged by the device as a command.

We are optimistic about the second problem, because the overall number of interrupts or the constraint on their intervals are could be reduced or eliminated, if we use more intelligent device controllers. Actually the old-fashioned controller employed in the machine on which we took these measurements, requires four interrupts for a SCSI command, but recent advanced SCSI controllers need only one interrupt. Our temporal solution to this problem is to delay the interrupt handling of the first phase, if the kernel is going to notify any occurrence of events.

3.2 Observations and Implementation Issues

We did not modify the UNIX server for our extensions. Because of this, management of realtime threads on Mach 3.0 is more complex than on Mach 2.5 when notification functions execute UNIX system calls. The kernel needs to propagate the realtime constraints to the UNIX server, so that specific threads of the UNIX server also meet the constraints. Since Mach IPC massaging has no out-of-band message and the messages are queued, it is impossible for the kernel to make only emergency threads run asynchronously in the UNIX server. In a multiple-server environment, this problem would be more serious; problems like priority inversion could occur. This issue is, however, not limited to Mach 3.0, but also occurs in Mach 2.5, when applications are coordinated with realtime and non-realtime threads. In this implementation, realtime threads and device handling threads, especially for Ethernet may have the same priority to avoid a deadlock. This results in causing unexpected delays. Figure 8 shows the problem caused by this implementation.

The number of AST for invocation to the preemptive deadline-scheduling was reduced due to the improvement of the context switching code(continuation code) in Mach 3.0(See "spin loop" and "fork" in Table 1). In spite of this improvement, the number of traps increased on Mach 3.0. This is because the UNIX server communicates with the kernel by Mach IPC, which could cause VM faults when transferring data. This could decrease the performance of other kernel activities, but at the same time provides more opportunities for the kernel to switch to a realtime thread.

4 Temporal Paging System

Advances in memory chips, processors, storage capacity, and data compression mean that multimedia applications that handle huge data segments will appear. To support continuous media in such environments, realtime access to virtual memory would be a serious problem. Because page fault handling causes unpredictable delays, specific regions of the address space need to be wired to physical memory pages. Mach 3.0 provide *vm_wire*() for this purpose; it wires down an address range in the task's map. Since it is, however, impractical to wire down such huge data segments, the availability is limited.

In addition, to meet realtime constraints, multimedia applications currently need to be implemented by exploiting knowledge of the hardware devices and the storage system; application programmers take a responsibility for the technology-mapping. As those devices progress, such applications need to modified to follow the advances.

Temporal Paging System(TPS) is designed to provide a technology-independent virtual memory access, including storage accesses to multimedia applications that have huge data segments and need realtime access to them. This feature was added to our extensions in the Mach 3.0 version, and is still in the experimental stage. The following subsection gives the abstractions in the temporal paging system.

4.1 Temporal and Attribute

"Temporal" is defined as special and volatile memory that exists only within a specific time duration $T \leq t < T + Q$. If ordinary memory is a one-dimensional array $memory[a]$ ($a = 0, 1, 2, \ldots$), then temporal memory occupies two-dimensional (address and time) space $temporal[a, t]$ ($T \leq t < T + Q$). The contents of memory overlapped by temporal memory can change temporally while the temporal memory is valid. After the duration, the original memory contents recover.

A region of temporal memory has an *attribute*, which has information on the initial value, the status after the valid duration, and which time the temporal memory is based on:

- Temporal memory has an initial value, $temporal[a, T]$. The initial value can be determined in advance, and the pointer to the source(memory and a specific device) is kept in the attribute. Because of this characteristic, temporal memory enables an application to update a specific region of memory at time intervals with no realtime operations on memory. Animated graphics is one example. The application only needs to determine $temporal[a, T + nQ](a \in A, n = 0, 1, 2, \ldots)$. A is the region of frame memory that includes the animated graphics. The status of the temporal after the valid duration depends on its attribute.

- Temporal memory after the valid duration becomes either memory or is removed from the address space, depending the attribute. In a virtual memory system, temporal memory is valid and identical to ordinary memory only within a specific time duration; before the duration, access to temporal memory causes an address error.

- Applications keep a system time and local time. The attribute indicates the time standard the temporal memory is based on.

Use of the complex number notation $temporal[z](z = a + it)$ gives the same interface to temporal memory as to memory (i is the imaginary number). For example, *temporal_t *malloc*(unsigned zsize) allocates a region of temporal memory and returns a pointer to it. See Figure 3.

4.2 Temporal Paging System

We combine temporal memory with a traditional paging system; we define "temporal page" as $temporal[a, t]$ ($M \leq a < M + P, T \leq t < T + Q$). TPS manages physical memory pages in the two-dimensional address and time space; a same physical memory page can be shared by multiple temporal pages, if they are not used simultaneously. Thus, the temporal paging system provides a more efficient management of physical memory pages and supports realtime access to virtual memory as well.

Discardable pages [13] can be implemented using temporal pages. Working memory used as a garbage collector does not need to be saved and restored by a pager if the work is completed. If the time needed for the work is predictable, the working memory should be allocated as temporal pages. The region will be automatically removed from the virtual address space after the work.

TPS provides an uniform interface for realtime access to virtual memory mapped to a specific device. Latencies for storage access vary with devices. To use physical memory efficiently, knowledge must be exploited not only of what data is needed but when it will

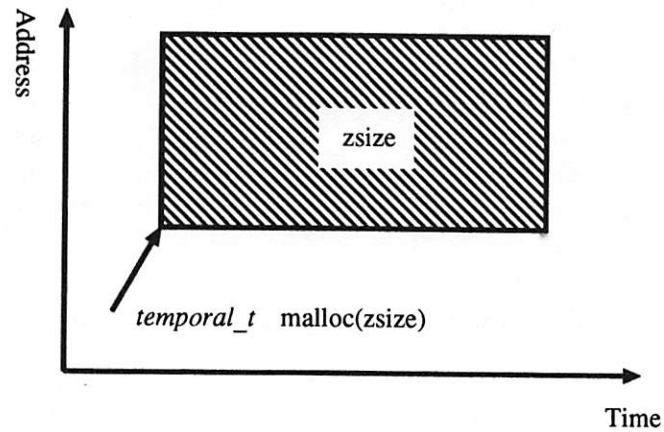


Figure 3: Temporal memory allocation using malloc()

be needed [11]. TPS provides knowledge of what and when the kernel should preload and swap out the data. The application programmers do not need to exploit knowledge of the hardware devices and the storage system.

The following primitives are partial interface to virtual temporal pages:

- `temporal_allocate()` — allocates a region of temporal pages
- `temporal_deallocate()` — deallocates a region of temporal pages
- `temporal_init_map()` — gives the initial value. The source is either memory or a specific device.
- `temporal_getstat()` — gets the attribute.
- `temporal_setstat()` — sets the attribute.

5 Conclusions and Lessons

To support multimedia applications that have realtime constrains, we have extended the Mach 2.5 and Mach 3.0 systems. Comparing the two implementation gave us lessons on the micro-kernel when implementing soft realtime systems. The main lessons are:

- Mach 3.0 has an advantage over Mach 2.5 when building realtime systems. Mach 2.5 needs more effort to reduce unexpected delays in the kernel.
- It is difficult, even for the Mach 3.0 kernel to respond in a realtime manner, when frequently executing I/O operations on devices, especially SCSI disks. This problem might be specific to the machine on which we took the measurements. A more intelligent SCSI controller could solve this problem. In addition, user-level device management is helpful when improving the device drivers.
- It is more complex to implement realtime systems in multiple-server environments. This issue is, however, not limited to Mach 3.0, but also occurs in Mach 2.5, when

applications are coordinated with realtime and non-realtime threads. We need to build realtime servers, otherwise system calls in realtime threads might be limited.

- To support multimedia applications that have huge data segments and realtime access to them, knowledge of what data is and when it will be needed must be exploited.

References

[1] M. J. Accetta, W. Baron, R. V. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. W. Young, "Mach: A New Kernel Foundation for UNIX Development," in *Proceedings of the Summer USENIX Conference*, July, 1986.

[2] David L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System," *IEEE Computer*, Vol.23, No.5, 1990.

[3] Eric C. Cooper and Richard P. Draves. "C Threads," Technical Report, Computer Science Department, Carnegie Mellon University, CMU-CS-88-154, March, 1987.

[4] Alssandro Forin, David Golub, and Brain Bershad, "An I/O System for Mach 3.0," in *Proceedings of USENIX Mach Workshop*, November, 1991.

[5] D. Golub, R. Dean, A. Forin, and R. Rashid, "Unix as an Application Program," in *Proceedings of the Summer USENIX Conference*, June, 1990.

[6] IEEE, "Realtime Extension for Portable Operating Systems," P1003.4/Draft10, February, 1991.

[7] C. L. Lui and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, Vol.20, No.1, 1973.

[8] Frank W. Miller, "Predictive Deadline Multi-Processing," *ACM Operating Systems Review*, Vol.24, No.4, October, 1990.

[9] Jun Nakajima, Masatomo Yazaki, and Hitoshi Matumoto, "Multimedia/Realtime Extensions for the Mach Operating System," in *Proceedings of the Summer USENIX Conference*, July, 1991.

[10] Lui Sha, Rajkumar, R., and Lehoczky, J.P., "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," Carnegie Mellon University, CMU-CS-87-181, 1987.

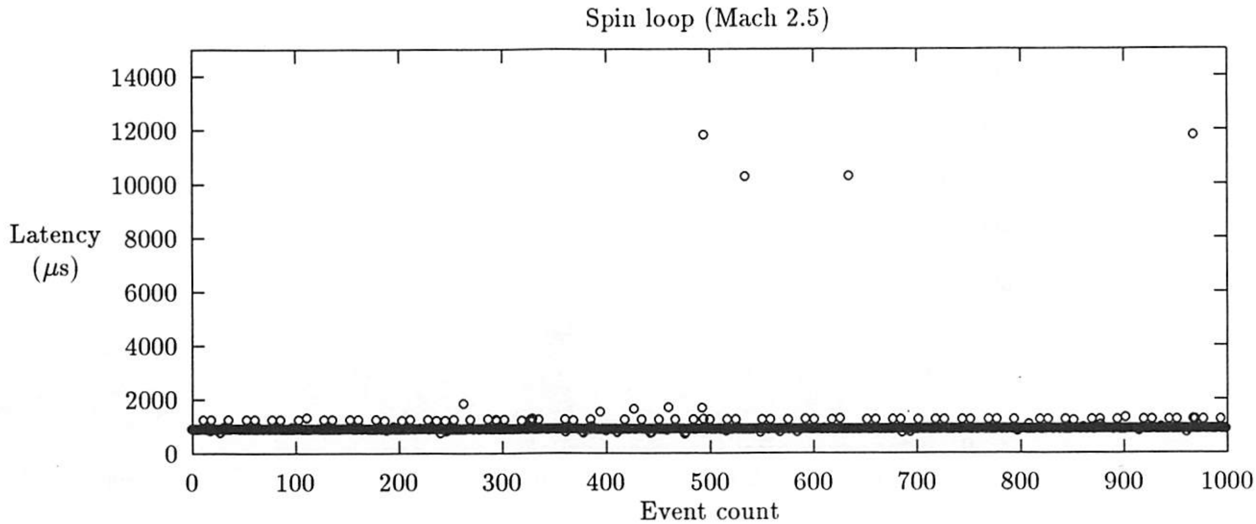
[11] Richard Staehli and Jonathan Walpole, "Constrained-Lantency Storage Access: A Survey of Application Requirements and Storage System Design Approaches," Technical Report, Deparment of Computer Science and Engineering, Oregon Graduate Institute, CS/E 91-019, October, 1991.

[12] John A. Stankovic and Krithi Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Operating Systems," *ACM Operating Systems Review*, Vol.23, No.3, July, 1989.

[13] Indira Subramanian, "Managing Discardable Pages with an External Pager," in *Proceedings of USENIX Mach Workshop*, November, 1991.

[14] J. S. Sventek, "An Architecture Supporting Multi-Media Integration," *IEEE Computer Society Office Automation Symposium*, Gaithersburg, MD, April, 1987.

[15] Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao, "Real-Time Mach: Towards a Predictable Real-Time System," in *Proceedings of USENIX Mach Workshop*, October, 1990.



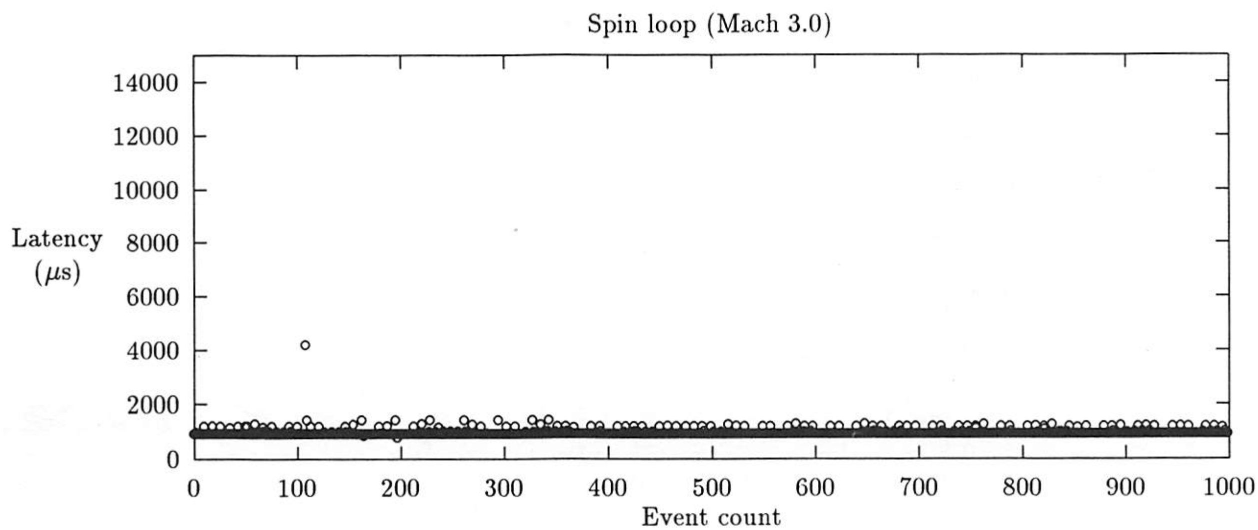


Figure 4: Event dispatch latencies when the "Spin loop" task is running

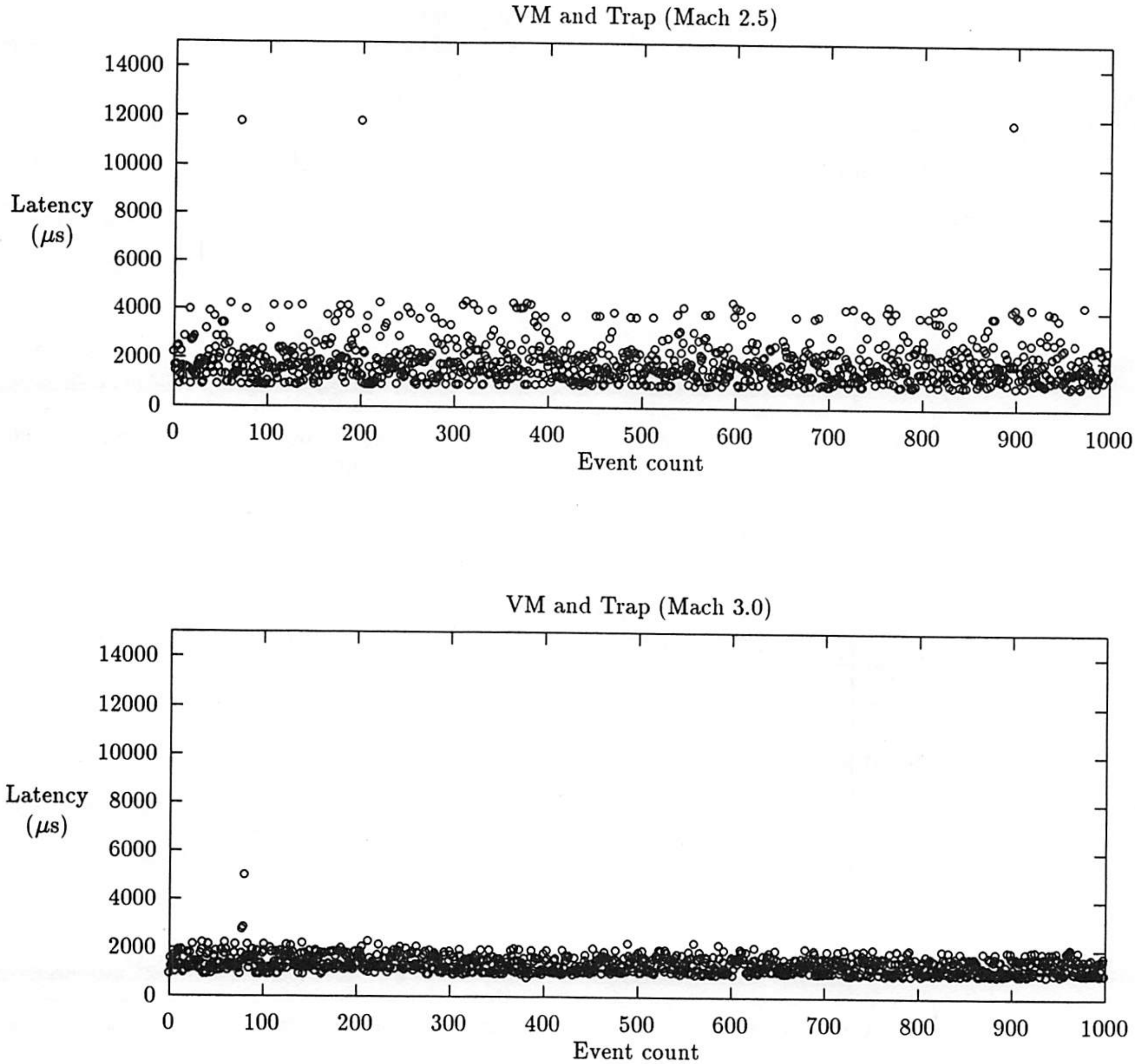
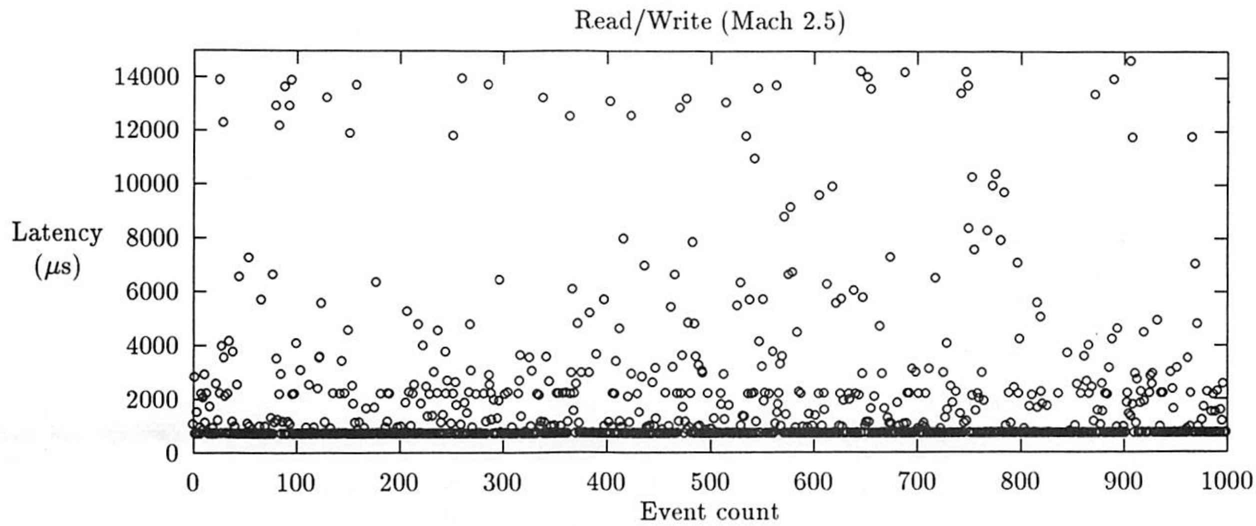


Figure 5: Event dispatch latencies when the "VM operation and trap" task is running



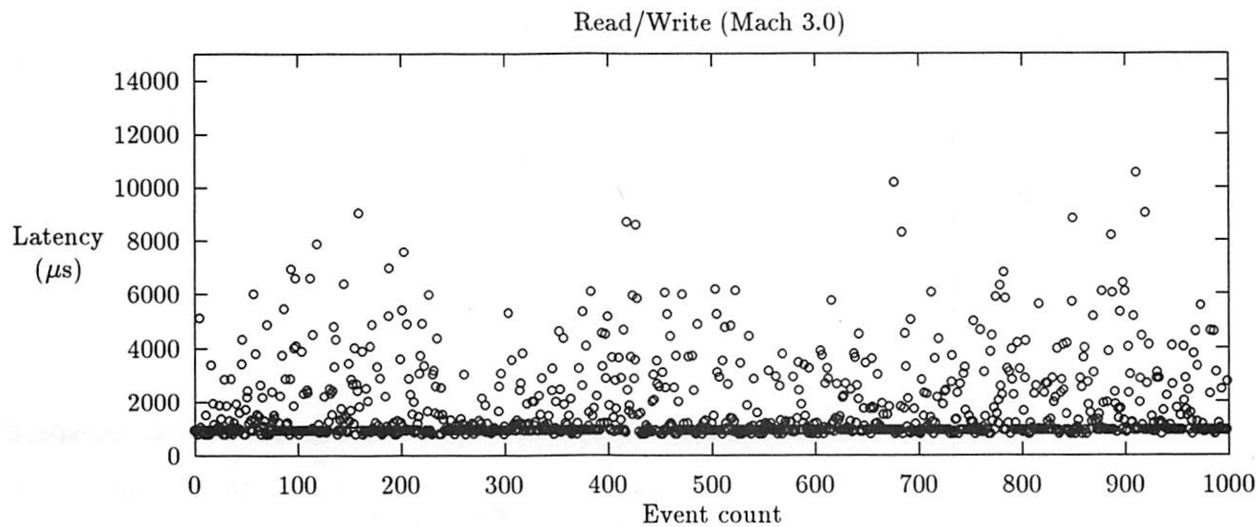
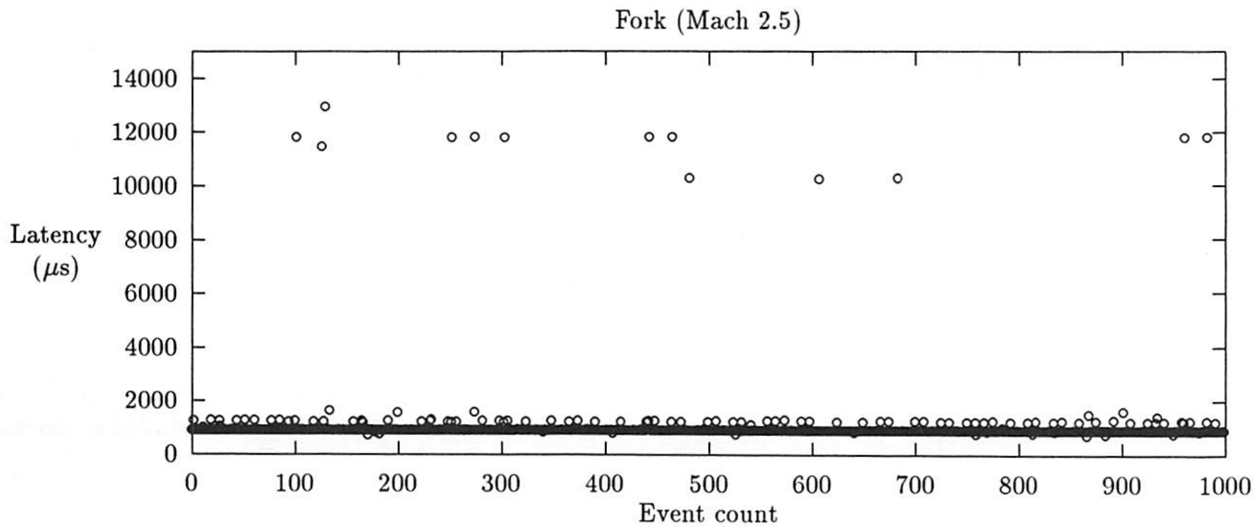


Figure 6: Event dispatch latencies when the " Read/write" task is running"



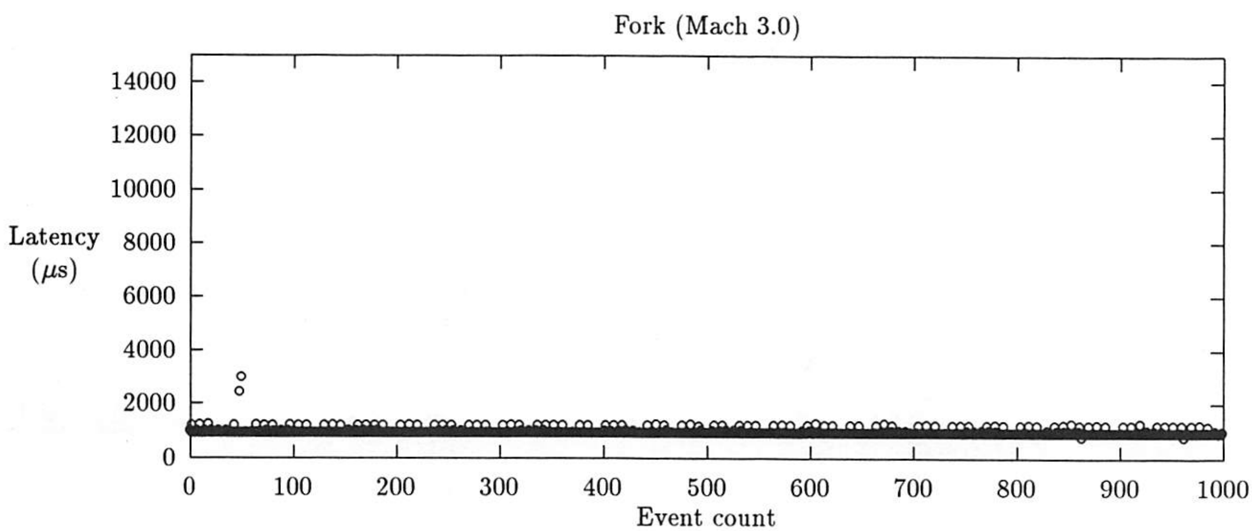


Figure 7: Event dispatch latencies when the "Fork operation" task is running"

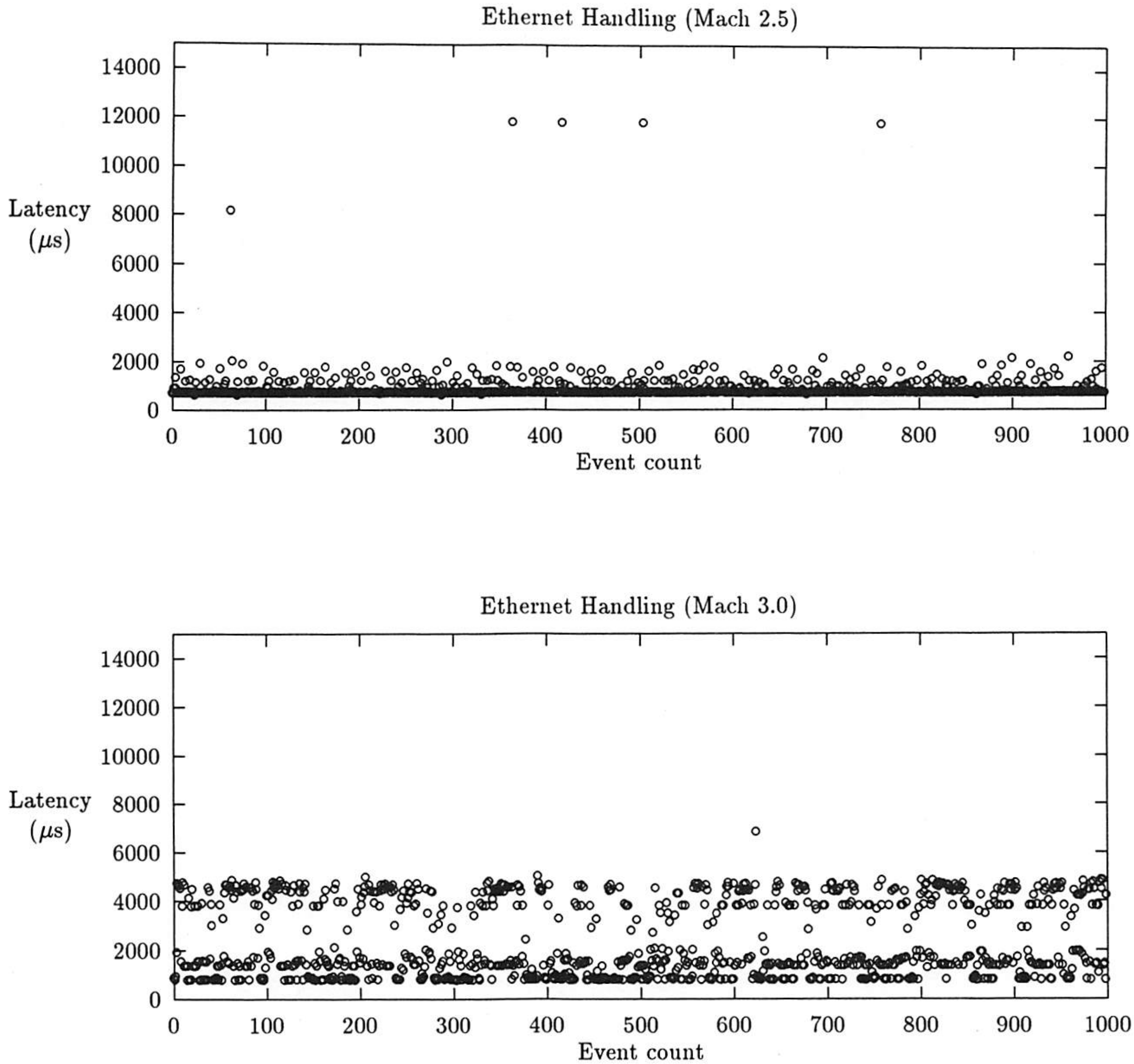


Figure 8: Event dispatch latencies when the "Ethernet handling" is running

Reimplementing the Synthesis Kernel on the Sony NeWS Workstation

Henry Massalin and *Calton Pu*[1]
Department of Computer Science
Columbia University
New York, NY 10027

calton@cs.columbia.edu

1 Introduction

Synthesis is an experimental operating system kernel that combines several new techniques to provide very high performance without sacrificing the expressive power or security of the system. The main ideas have been described in our other papers [2, 3, 4, 6, 7], including:

- *Run-time code synthesis* [7] — a systematic way of creating executable machine code at runtime to optimize frequently-used kernel routines for specific situations, greatly reducing the execution time for these calls.
- *Fine-grain scheduling* [3] — a process-scheduling technique based on the idea of feedback that performs frequent scheduling actions and policy adjustments (at sub-millisecond intervals) resulting in an adaptive, self-tuning system that can support processing of real-time data streams.
- *Lock-free optimistic synchronization* [4] to further reduce overhead and increase concurrency within the multi-threaded Synthesis kernel.

These ideas are tied together using a strongly modular "building-block" approach [6] in kernel design. This modularity facilitates a graceful extension of kernel services and additional support of hardware devices without accumulating undesirable overhead. The result is a significant performance improvement over traditional operating system implementations in addition to providing services.

Like many substantial software systems, Synthesis has been rewritten several times. Most recently, the kernel has been ported to a Sony NeWS workstation. At the same time, new functionality was added and its structure improved. This paper reports on the experience of porting and re-implementing the Synthesis kernel. In addition, we also discuss some issues in the impact of multiprocessor management and virtual memory management on the new Synthesis kernel.

2 The Environment

2.1 Hardware Base

At the time of this writing, Synthesis runs on two machines: the Sony NeWS 1860 workstation and a home-brew experimental computer, called the *Quamachine*. The Quamachine is

[1]This work was partially supported by National Science Foundation, IBM Corp., AT&T Foundation, Digital Equipment Corp., SUN Microsystems, and Sony Computer Science Lab.

a 68030-based single-board computer system designed to aid systems research and measurement. Measurement facilities include an instruction counter, a memory reference counter, hardware program tracing, and an interval timer with 20-nanosecond resolution. Other features of the Quamachine include 256 kilobytes of no-wait-state ROM that holds the entire Synthesis kernel, monitor, and runtime libraries; $2\frac{1}{2}$ megabytes of no-wait-state main memory; and audio I/O devices: stereo 16-bit analog output, stereo 16-bit analog input, and a compact disc (CD) player digital interface.

The Sony NeWS 1860 is a workstation with two 68030 processors. It is a commercially available machine, making Synthesis potentially accessible to other interested researchers. While two is not a large number, the Sony workstation demonstrates the Synthesis support for multiprocessing. While its architecture is not symmetric — one processor is the main processor and the other is the I/O processor — Synthesis treats it as if it were a symmetric multiprocessor, scheduling tasks on either processor without preference, except those that require I/O access that is accessible only from the I/O processor.

The Quamachine version of the Synthesis kernel was developed incrementally over several years. As the Synthesis ideas matured, we decided to reimplement the kernel on the Sony workstation in a systematic effort to clean up the artifacts of an incremental development.

2.2 The Macro Assembler

Synthesis is written in the 68030 assembly language with our own macro facility. Despite its obvious flaws — the lack of portability and the difficulty of writing complex programs in it — the assembly language was chosen because no high level programming language was found that provided both efficient execution and support for run-time code generation. The macro assembly language was for us a fast prototyping language. This may sound peculiar, since usually people use high level programming languages for fast prototyping. But in our case, assembler was beneficial because much of the early development work involved discovering the most efficient way of working with the machine and its devices. Also, the Synthesis assembler (written in C, by the way) assembles 5000 lines of code per second on the Quamachine. Complete system generation takes only 15 seconds. We are much more likely to try different design alternatives when the turnaround time is so short.

A powerful macro facility was developed to minimize the difficulty of writing complex programs. The Synthesis assembler macro processor borrows heavily from the C-language macro processor, sharing much of the syntax and semantics. But it provides some important extensions, including macros that can define macros and quoting and "eval" mechanisms capable of generating executable code dynamically.

The Synthesis kernel building blocks are called *quajects*. Quaject definition is a declarative macro-instruction in the assembly language. This kind of macros creates all the code and data structures needed by the kernel code generator, so the programmer no longer must worry about these details and can concentrate on the quaject's algorithms. Code written using these macro packages looks more like a higher-level language than assembler. We believe the insight gained can help guide research into new high level programming languages that support efficient run-time code generation.

We also felt that it would be an interesting experiment to write a medium-size system in assembler, which allows unrestricted access to the machine's architecture, and perhaps discover new coding idioms that have not yet been captured in a higher-level language. Surprisingly, we found that there are some things that were actually *easier* to do using Synthesis assembler than using C [1]. One example is the finite state machines found in various I/O drivers and protocol engines. Even the conceptually simple function of terminal emulation gives rise to difficult problems when coded in C. For example, how do we handle the case when the input buffer runs out while in the middle of parsing a complex escape sequence? Saving the data-part of the state is easy. But recording the context in which

processing stopped is much harder because C does not allow direct access to the program counter. This problem does not arise in assembly language.

2.3 The Synthesis Kernel Monitor

The Synthesis kernel monitor is a C-language parser front-end with direct access to the entire physical memory. The monitor runs as a kernel thread under Synthesis, allowing it full access to all the machine's resources and kernel code generators. It was crucial to both the development and porting of Synthesis because it let us run and test kernel quajects without requiring the full kernel to be functional.

When the monitor is started, it creates a thread, which executes as directed by commands typed at the monitor prompt. Commands include the usual complement typically found in debuggers, such as memory fill and modify. Commands also include arbitrary C-language statements, which are parsed into an intermediate representation and passed onto the kernel code generator. The parser also recognizes a number of special names, which refer to the monitor thread's internal state such as the CPU registers.

3 The Reimplementation Process

The Synthesis reimplementation happened in three stages. First, a minimal Synthesis kernel was ported to run under Sony's native UNIX as a process. Then we wrote drivers for the keyboard and screen, and got minimal Synthesis kernel to run on the raw hardware. This was followed by a full port, including all the device drivers, multiprocessor management, and virtual memory management.

3.1 The Quamachine Kernel

The Synthesis kernel is designed to support a real, full-featured operating system with functionality on the level of UNIX and Mach. It is built out of many modules called quajects. A quaject is a collection of code and data with a well-defined interface that performs a specific function and makes no calls to external code except through its interface. Kernel quajects include various kinds of queues and buffers, threads, TTY input and output editors, terminal emulators, and text and graphics windows. All higher-level kernel services are created by instantiating and linking two or more quajects. For example, a UNIX-like TTY device is built using the following quajects: A raw serial device driver, two queues, an input editor, an output format converter, and a system call dispatcher.

The wide choice of quajects and linkages allows Synthesis to support a wide range of different system interfaces at the user level. For example, Synthesis includes a partial UNIX emulator that runs a limited number of SUN-3 binaries. (The limitation is imposed by the incomplete Synthesis implementation, not the interface.) At the same time, different applications might use different interfaces, for example, one that supports asynchronous I/O.

The kernel on the Quamachine had several shortcomings. While the kernel showed impressive speed gains over conventional operating systems such as UNIX, its internal structure was not clean. The quaject structuring idea came late in kernel development, so there were many parts that had been written in an *ad hoc* manner. Furthermore, the Quamachine kernel did not support virtual memory or networking.

The goals of the Synthesis port to the Sony workstation was to alleviate the shortcomings, for example, by cleaning up the kernel structure and adding virtual memory and networking support. In particular, we wanted to show that the additional functionality would not significantly slow down the Synthesis kernel.

3.2 Minimal Kernel under Emulation

The first step went fast, taking two to three weeks. The reason is that most of the quajects do not need to run in kernel mode in order to work. The difference between Synthesis under UNIX and native Synthesis is that instead of connecting the final-stage I/O quajects to I/O device driver quajects (which are the only quajects that must be in the kernel), we connect them to UNIX `read` and `write` system calls on appropriately opened file descriptors. This is a solid proof that Synthesis services can run in user-level as well as kernel.

3.3 Minimal Kernel on Bare Machine

Porting to the raw machine was much harder, primarily because we chose to write our own device drivers. Some problems were caused by incomplete documentation on how the I/O devices on the Sony NeWS workstation work. It was further complicated by the fact that each CPU has a different mapping of the I/O devices onto memory addresses and not everything is accessible by both CPUs. A simple program was written to patch the running UNIX kernel and install a new system call — "execute function in kernel mode." Using this utility (carefully!), we were able to examine the running kernel and discover a few key addresses. After a bit more poking around, we discovered how to alter the page mappings so that sections of kernel and I/O memory were directly mapped into all user address spaces.[2] (The `mmap` system call on `/dev/mem` did not work.) Then using the Synthesis kernel monitor running on minimal Synthesis under a UNIX process, we were able to "hand access" the remaining I/O devices to verify their address and operation.

But the most difficult porting problems were caused by timing sensitivities in the various I/O devices. Some devices would "freeze" when accessed twice in rapid succession. These problems never showed up in the UNIX code because UNIX encapsulates device access in procedures. Calling a procedure to read a status value or change a control register allows enough time for the device to "recover" from the previous operation. But with code synthesis, device access frequently consists of a single machine instruction. Often the same device is accessed twice in rapid succession by two consecutive instructions, causing the timing problem. Once the cause of the problem was found, it was easy to correct: we made the kernel code generator insert an appropriate number of "nop" instructions between consecutive accesses.

3.4 The New Kernel

Once we had the minimal kernel running, getting the rest of the kernel and its associated libraries working was relatively easy. All of the code that did not involve the I/O devices ran without change. This includes the user-level shared runtime libraries, such as the C functions library and the signal-processing library. It also includes all the "intermediate" quajects that do not directly access the machine and its I/O devices, such as buffers, symbol tables (for name service), and mappers and translators (for file system mapping). Code involving I/O devices was harder, since that required writing new drivers. Finally, there are some unfinished drivers such as the SCSI driver.

The thread system needed some changes to support the two CPUs on the Sony workstation. Most of the changes were in the scheduling and dispatching code, to synchronize between the processors. This involved developing efficient, lock-free data structures which were then used to implement the algorithms. While doing these changes, we also changed the scheduling policy from a single round-robin queue to one that uses a multiple-level queue structure. This helped guarantee good response time to urgent events even when there are many threads running, making it feasible to run up to thousands of threads on Synthesis.

[2] Talk about security holes!

The most time-consuming part was implementing the new services: virtual memory, ethernet driver, and window system. They were all implemented "from scratch", using all the performance-improving ideas used in the rest of Synthesis, such as kernel code generation. Recent measurements [1] show similar performance gains in these areas as well. The ethernet driver, for example, is fast enough to record all the packet traffic of a busy ethernet (400 kilobytes/second) into RAM using only 20% of a 68030 CPU time. This is a problem that has been worked on and dismissed as impractical except when using special hardware.

Besides the Sony workstation, the new kernel runs on the Quamachine as well. Of course, each machine must use the appropriate I/O drivers, but all the new services added to the Sony version work on the Quamachine.

4 Some Processor Management Issues

Although both the Quamachine and the Sony workstation have 68030 CPUs, there are also some significant differences. For example, the Sony workstation has two CPUs, thus it supports multiprocessing, discussed in Section 4.1. In contrast, the Quamachine supports more complete control of the floating point co-processor, discussed in Section 4.2. While broad, system-level performance measurements have not yet been made, we have made operation-level measurements that can be compared with previous versions of Synthesis. These generally reveal performance nearly identical to that previously achieved on the Quamachine, adjusted for clock rate.

4.1 Thread Scheduling and Dispatching

The introduction of the lock-free synchronization code for multiprocessor support has slowed down some thread operations, most notably, scheduling and dispatching. The original kernel used a very fast, executable data structure implementation of the ready queue, leading to very fast context switch times of between 7 to 50 microseconds, depending on how much state was being switched.[3] Lock-free synchronization, even though shown to be faster than conventional locking techniques [1], carries more overhead than the original implementation based on executable data structures.

Table 1 summarizes the time taken by the various types of context switches in Synthesis, saving and restoring all the integer registers. These times include the hardware interrupt service overhead — they show the elapsed time from the execution of the last instruction in the suspended thread to the first instruction in the next thread. Previously published papers report somewhat lower figures [7] [2]. This is because they did not include the interrupt-service overhead, and because of some extra overhead incurred in handling the 68882 floating point unit on the Sony NeWS workstation that does not occur on the Quamachine, as discussed later. For comparison, a call to a null procedure in the C language takes 1.4 microseconds, and the Sony UNIX context switch takes 170 microseconds.

4.2 Floating Point

The Quamachine has some special hardware that enable and disables the floating point unit by software command, allowing faster context switching of threads that did not use floating

[3] Previous papers incorrectly cite a floating point context switch time of 15 μS [7] [2]. This error is believed to have been caused by a bug in the Synthesis assembler, which incorrectly filled the operand field of the floating point move-multiple-registers instruction causing it to preserve just one register, instead of all eight. Since very few Synthesis applications use floating point, this bug remained undetected for a long time.

Type of context switch	Time (μS)
Integer registers only	12
Floating-point	52
Integer, change address space	17
Floating-point, change address space	56
Null procedure call (C langauge)	1.4
Sony NeWS, UNIX	170

68030 CPU, 25MHz, 1-wait-state main memory, cold cache

Table 1: Cost of Thread Scheduling and Context Switch

point since they were last switched in. The Sony machine lacks this hardware, which added 1.5 μS to every thread's context-switch times. Switching floating point context is expensive because of the large amount of state that must be saved. The registers are 96 bits wide; moving all eight registers requires 24 transfers of 32 bits each. The 68882 coprocessor in the Sony workstation compounds this cost, because each word transferred needs two bus cycles: one to fetch it from the coprocessor, and one to write it to memory. The result is that it takes about 50 microseconds just to save and restore the hundred-plus bytes of information comprising the floating point coprocessor state. This is more than five time the cost of doing an entire context switch without the floating point.

Since preserving floating point context is so expensive, we use runtime tests to see if floating point had been used to avoid saving state that is not needed. Threads start out assuming floating point will not be used, and their context-switch code is created without it. When context-switching out, the context-save code checks whether the floating point unit had been used. It does this using the `fsave` instruction of the Motorola 68882 floating point coprocessor, which has been designed with this purpose in mind [5]. If there was floating point activity, the floating point state is saved, and the context-switch code re-created to include the floating point context in subsequent context switches. Since the majority of threads in Synthesis do not use floating point, the savings are significant.

Unfortunately, after a thread executes its first floating point instruction, floating point context will have to be preserved from that point on, even if no further floating point instructions are issued. The context must be restored upon switch-in because a floating point instruction might be executed. The context must be saved upon switch-out even if no floating point instructions had been executed since switch-in because the 68882 cannot detect a lack of instruction execution. It can only tell us if its state is completely null. This is bad because sometimes a thread may use floating point at first, for example, to initialize a table, and then not again. But with the 68882, we can only optimize the case when floating point is never used.

The Quamachine has hardware to alleviate the problem. Its floating point unit — also a 68882 — can be enabled and disabled by software command, allowing a lazy-evaluation of floating point context switches. Switching in a thread for execution loads its integer state and disables the floating point unit. When a thread executes its first floating point

System Activity	Kernel Memory Use (Kbytes)
Boot image for full kernel	140
One thread running	Boot + 8
File system and disk buffers	Boot + 400
100 threads, 300 open files	Boot + 1400

Table 2: Kernel Memory Requirements

instruction since the switch, it takes an illegal instruction trap. The kernel then loads the necessary state, first saving any prior state that may have been left there, reenables the floating point unit, and the thread resumes with the interrupted instruction. The trap is taken only on the first floating point instruction following a switch, and adds only 3 μS to the overhead of restoring the state. This is more than compensated for by the other savings: integer context-switch becomes 1.5 μS faster because there is no need for an **fsave** instruction to test for possible floating point use; and even floating point threads benefit when they block without a floating point instruction being issued since they were switched in, saving the cost of restoring and then saving that context. Indeed, if only a single thread is using floating point, the floating point context is never switched, remaining in the coprocessor.

5 Some Memory Management Issues

Another area requiring attention is memory management. This is a particular concern during the reimplementation because we have added virtual memory support.

5.1 Kernel Size

Kernel size inflation is an important concern in Synthesis due to the potential redundancy in the many functions generated at runtime. This could be particularly bad if layer collapsing — a technique where runtime-generated functions are in-line substituted into other functions — were used too enthusiastically. To limit memory use, Synthesis can generate either in-line code or subroutine calls to shared code. The decision of when to expand in-line is made by the programmer writing the code generator. Full, memory-hungry in-line expansion is usually reserved for specific uses where its benefits are greatest: the performance-critical, frequently-executed paths of a function, where the performance gains justify increased memory use. Less frequently executed parts of a function are stored in a common area, shared by all instances through subroutine calls.

In-line expansion does not always cost memory. If a function is small enough, expanding it in-line can take the same or less space than calling it. Examples of functions that are small enough include character-string comparisons and buffer-copy. For functions with many runtime-invariant parameters, the size expansion of inlining is offset by a size decrease that comes from not having to pass as many parameters.

In practice, the actual memory needs are modest. Table 2 shows the total memory used by the full Sony kernel — including I/O buffers, virtual memory, network support, and a window system with two memory-resident fonts.

5.2 Protecting Synthesized Code

The classic solutions used by other systems to protect their kernels from unauthorized tampering by user-level applications also work in the presence of synthesized code. Kernel data and code — both synthesized and not — are protected using memory management to make the kernel part of each address space inaccessible to user-level programs. Synthesized routines run in supervisor mode, so they can perform privileged operations such as accessing protected buffer pages.

User-level programs enter supervisor mode using the `trap` instruction. This instruction provides a controlled — and the only — way for user-level programs to enter supervisor mode. The synthesized routine implementing the desired system service is accessed through a jump table in the protected area of the address space. The user program specifies an index into this table, ensuring the synthesized routines are always entered at the proper entry points. This protection mechanism is similar to Hydra's use of C-lists to prevent the forgery of capabilities [8].

Once in kernel mode, the synthesized code handling the requested service can begin to do its job. Further protection is unnecessary because, by design, the kernel code generator only creates code that touches data the application is allowed to touch. For example, were a file inaccessible, its `read` procedure would never have been generated. Just before returning control to the caller, the synthesized code reverts to the previous (user-level) mode.

5.3 Non-coherent Instruction Cache

A common assumption in the design of processors is that a program's instructions will not change as the program runs. For that reason, most processor's instruction caches have no coherency control — writes to main memory are not reflected in the cache. Runtime code generation violates this assumption, requiring that the instruction cache be flushed whenever new code is generated. In a shared-memory multiprocessor, the caches in all the CPUs need to be flushed. In the Sony workstation, this is done by sending an interrupt to the other CPU, asking it to flush its cache. However, too much cache flushing reduces performance, both because programs execute slower when the needed instructions are not in cache and because flushing itself may be an expensive operation.

Fortunately, in the majority of code synthesis applications, an incoherent cache is not a big problem. The cost of flushing even non-local caches contributes relatively little compared to the cost of allocating memory and creating the code. On the NeWS 1860, for example, it takes 2.5 μS elapsed time to process the cache-flush interrupt. In comparison, it takes 12 μS to create a buffer, 20 μS to create a thread, and 71 μS to create a text window. So while cache flushing adds as much as 20% to the cost of creating simple pieces of code, it is not inordinately large. If code generation happens infrequently relative to the code's use, as is usually the case, the performance hit is small.

The problem would have been much more severe if the generated code were self-modifying. The performance of self-modifying code, like that found in executable data structures, suffers much from an incoherent instruction cache. This is because the ratio of code modification to use tends to be high. Ideally, we would like to flush with cache-line granularity to avoid losing good entries. Fortunately, this is possible on the 68030 processor. But even line-at-a-time granularity has its disadvantages: it needs machine registers to hold the parameters, registers that may not be available during interrupt service without incurring the cost of saving and restoring them. In practice, most cases of self-modifying code actually occur inside interrupt service routines where small amounts of data (e.g., one character for a TTY line) must be processed with minimal overhead. Fortunately, in all important cases the cache invalidation cost has been reduced to zero through careful layout of the code in memory using knowledge of the 68030 cache architecture to cause the subsequent instruction

fetch to replace the cache line that needs flushing. Since only the I/O processor can service these interrupts, the main processor's cache would never contain the modified entries and need not be flushed. But this trick is neither general nor portable.

6 Evaluation

The initial applications envisioned for Synthesis were real-time signal processing and parallel computation. Before the port, the Quamachine kernel supported fast kernel threads and a rich set of I/O devices and operations, including real-time audio. With the port came new functionality, including:

- Virtual memory.
- Efficient multiprocessor support using lock-free optimistic synchronization.
- An ethernet network driver.
- An efficient, extensible window system based on quajects.

All these services were added without slowing down the basic system services already there. This experience validates the Synthesis approach in the use of runtime code generation to allow expansion of kernel services while avoiding the problem of mounting overhead that plague existing operating systems. In particular, we find that the addition of virtual memory support does not slow down the other basic services such as threads when the virtual memory services are not being used. Thread context switch time varies from 12 microseconds for a same-address-space, non-floating point switch to 56 microseconds for a full context switch, including changing address spaces plus saving and restoring the floating point state.

All the tools described in Section 2 were instrumental in the development and evaluation of both the Quamachine version and the Sony version of the system. The macro assembler provides run-time code generation facilities; the kernel monitor facilitates testing and debugging; the hardware assist on the Quamachine makes detailed measurements possible. In a "normal" system without these tools, it would have been very difficult to develop a kernel like Synthesis.

Another good experience is the gradual port of the Synthesis kernel in three stages. The modularization of the kernel in terms of quajects and the tools made the gradual reimplementation feasible. At each stage we only had to deal with a limited set of debugging and design problems. As the kernel is refined, more functionality is added without sacrificing the performance or modularity.

References

[1] H. Massalin. *Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, Forthcoming 1992.

[2] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 191–201, Arizona, December 1989.

[3] H. Massalin and C. Pu. Fine-grain adaptive scheduling using feedback. *Computing Systems*, 3(1):139–173, Winter 1990. Special Issue on selected papers from the Workshop on Experiences in Building Distributed Systems, Florida, October 1989.

[4] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Department of Computer Science, Columbia University, February 1991.

[5] Motorola. MC68881 and MC68882 Floating-Point CoprocessorUser's Manual Prentice Hall, Englewood Cliffs, NJ, 1987

[6] C. Pu and H. Massalin. Quaject composition in the synthesis kernel. In *Proceedings of International Workshop on Object Orientation in Operating Systems*, Palo Alto, October 1991. IEEE/Computer Society.

[7] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[8] W.A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessing operating system. *Communications of ACM*, 17(6):337–345, June 1974.

A Model and Prototype of VMS Using the Mach 3.0 Kernel

Cheryl A. Wiecek
Christopher G. Kaler
Stephen Fiorelli
William C. Davenport, Jr.
Robert C. Chen

Digital Equipment Corporation
110 Spit Brook Road
Nashua, NH 03062-2698
{wiecek,kaler,fiorelli,davenport,rchen} @star.enet.dec.com

Abstract

Digital's VMS operating system has been a successful software base for our VAX processors since the late 1970's. Existing operating systems are facing many new requirements and challenges in the 1990's and beyond. This has led us to investigate new approaches for designing, implementing, and maintaining VMS. One such effort is described in this paper. Using the Mach 3.0 kernel from Carnegie Mellon University, we developed a multi-server model and prototype of VMS that emphasizes platform-independence and internal partitioning. We describe the challenges we faced, the model that we evolved to address these challenges, and the prototype that we built to demonstrate feasibility. We conclude with a discussion of our findings and possible future directions.

1 Introduction

VMS[1] (Virtual Memory System) is an operating system that was designed starting in 1976 as a general-purpose time sharing system for Digital's VAX line of processors. Over time, VMS [7] has evolved into a large collection of functions and features that support a large customer base and many applications. Preserving and making enhancements to VMS is a challenging effort in light of its dependencies on the VAX architecture and its monolithic structure. This has motivated us to investigate ways to restructure VMS to accommodate open and distributed computing requirements in and beyond the 1990's while continuing to support current VMS customers and applications.

Mach [1,9,11] includes base support for architecture-independent virtual memory management, threads, and interprocess communication. Whether or not Mach could support VMS was intriguing to both CMU and Digital. The micro-kernel [5] approach that Mach 3.0 represents also had potential for addressing architectural dependencies and structural issues facing VMS. The possibility of building a VMS environment on top of Mach using a number of servers that might be shared, replicated, and replaced as needed was promising.

After studying Mach literature sent to us by CMU, a VMS engineering advanced development effort was proposed to investigate the potential of micro-kernel technology to support a VMS environment using Mach 3.0. The goals were:

- Learn the requirements that VMS has on a micro-kernel.
- Understand the strengths and weaknesses of a micro-kernel approach for VMS.

[1] VMS, VAX, VAXstation, VAXcluster, and Digital are trademarks of Digital Equipment Corporation.

- Identify features, mechanisms, and policies in VMS that are difficult to port to non-VAX architectures and suggest alternatives.
- Investigate the implications of a micro-kernel approach for VMS-based applications.

Porting VMS to a non-VAX platform or developing a product version of VMS based on the Mach 3.0 kernel was not part of this research effort.

A small team of VMS engineers was assembled by January of 1991 and is working to demonstrate that it is both possible and desirable to build an operating system that:

- uses minimal kernel and client-server design techniques to ease change and evolution,
- promotes portability with an implementation that emphasizes architectural independence,
- has respectable performance,

  and
- preserves the VMS user environment.

An agreement with CMU gave us access to the expertise of, and support from, the CMU Mach team. One of the VMS engineers on the project relocated to Pittsburgh, PA. and worked from the CMU campus.

The project evolved into a two-phase effort: developing a model of VMS using the Mach 3.0 kernel and design philosophy, and implementing prototype software based on the model.

During the first phase of the project, we developed a model of VMS that isolates the VAX architectural dependencies and existing VMS software interdependencies. This phase lasted about eight months. The model we produced is our vision of how VMS could be structured to ease evolution, portability, and maintenance. In this model, we partition VMS into multiple platform-independent servers and use a client-server design approach to provide VMS user processes with access to this VMS environment. Each VMS process consists of two cooperating Mach tasks: a user image space where VMS process images execute, and a process server that manages the VMS process and its images.

During the second phase of the project, we developed prototype software that includes six servers to demonstrate the viability of our model of VMS. This prototype runs on a VAXstation and provides for execution of existing non-privileged VMS binary images. VMS server tasks include a library that provides access to functions in the Mach 3.0 kernel and 4.3 BSD UNIX[1]-compatible single server that we obtained from CMU. We are currently measuring and evaluating this prototype.

In the rest of this paper, we describe our model and the prototype work we have completed. This includes details on the challenges and issues we faced, as well as our model design, prototype implementation, and findings.

2 VMS Model Development

A major undertaking for the team was to come up with a model of VMS that addressed the goals and objectives we set for the project. The model we developed does not cover all aspects of VMS. In this section we describe the challenges we addressed and present some details on the model we designed. In particular, we discuss the VMS process, the set of VMS servers that create the VMS operating system (personality) environment, and a number of VMS mechanisms affected by our approach.

2.1 Challenges

An important milestone for the project was to understand the VMS personality environment and to develop a model of VMS that includes the functions, policies, and mechanisms needed to support the execution of VMS non-privileged (user) images.

[1]UNIX is a registered trademark of UNIX System Laboratories, Inc.

There were two issues that we wanted to address with this model: VMS dependencies on the VAX architecture and interdependencies between VMS subsystems. Had we chosen to address just VMS dependencies on the VAX architecture, the model would focus on replacing VMS kernel primitives with Mach 3.0 primitives, leaving the rest of VMS alone (a single-server approach). To deal with interdependencies between VMS subsystems as well, we chose to pursue a multi-server approach where the VMS operating system is partitioned into a set of servers that act on behalf of user (client) processes.

Our multi-server approach raised some interesting issues with regard to data sharing within VMS, and between VMS and user processes (including protection of certain data from access by user processes). Moving VMS subsystems into separate address spaces invalidated the shared memory assumptions made by the existing subsystems. Due to the way the VAX architecture partitions virtual memory, alternatives were needed to handle:

- unprotected data in process (P0) address space that is accessed by VMS subsystems.
- protected VMS subsystem data that currently resides in process (P1) address space. VMS subsystems, executing under process context, have access to per-process data areas in P1 space.
- protected data areas shared among VMS subsystems in the shared system (S0) address space where VMS is itself located. There are many control-block-based structures in VMS that contain pointers to other structures and affect multiple VMS subsystems.

In addition, we did not want to assume, as VMS does, that four access modes are available to protect data.

Our model impacted a number of mechanisms involved in the execution of a VMS process. Some of these mechanisms depend on VAX architecture features, whereas others are affected by our multi-server approach. These include:

- asynchronous system traps (ASTs),
- exception and condition handling,
- interrupts, interrupt priority level (IPL), and spinlocks,

  and
- scheduling-related mechanisms (*e.g.*, I/O completion, event flag wait).

We also wanted to consider the implications of multiple operating system personalities on one kernel and the possible interoperability that could be exploited. One example is allowing for multiple command language interpreters (*e.g.*, VMS DCL, POSIX [8], DEC/Shell, *etc.*). Another is investigating generic servers that multiple personalities might share (or at least partitioning servers into personality-independent and personality-dependent pieces).

2.2 VMS Model Design

In addressing the challenges just discussed, our model of VMS has the following features:

- The VMS process, called the user client, consists of a pair of Mach tasks. These two tasks provide user images with a VMS application environment.
- This user client is partitioned into operating system personality-independent and personality-dependent pieces.
- VMS servers run in user mode and do not depend on processor architectural features.
- User-generated VMS actions (like setting an event flag or delivering an AST) are implemented in an architecture-independent manner.

An overview of the model and some details on these features are presented below.

2.2.1 Overview of the Model

The VMS-on-Mach model consists of two basic types of tasks: VMS server tasks and VMS user space tasks, all depicted using circles in Figure 1. The VMS server tasks provide the VMS personality environment that runs on the Mach 3.0 kernel. Services provided by this environment are used by VMS user images executing within a user space task.

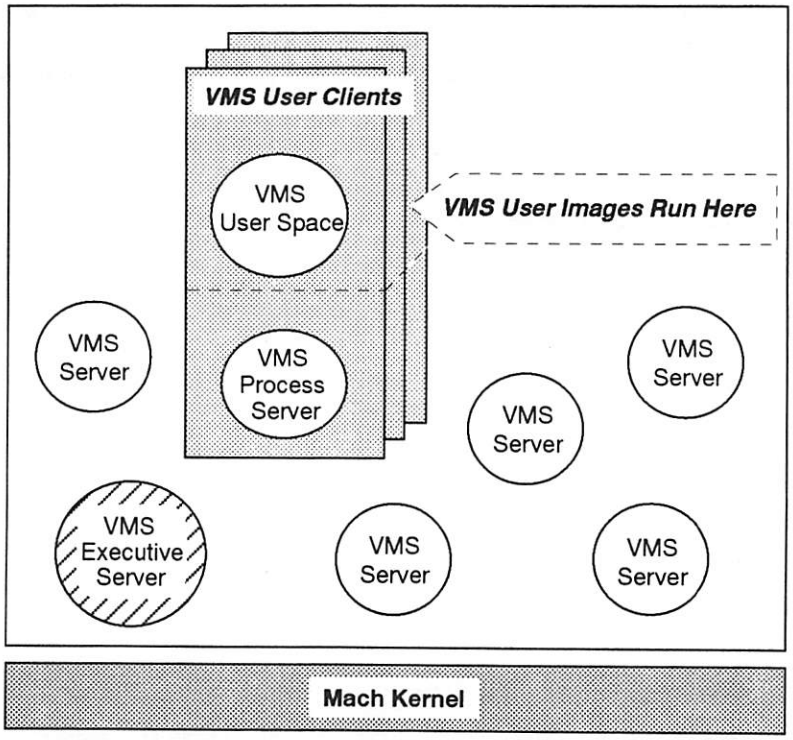


Figure 1 VMS-on-Mach Model

Each VMS process (user client), depicted by a shaded rectangle in Figure 1, consists of a VMS user space task and a VMS process server task. When system service requests are made by VMS user images, these are translated within the VMS user space task into requests that are acted upon by that task or, using Mach IPC (interprocess communication) services, by other VMS server tasks. The VMS process server task has special responsibility for providing the address space within which process-protected code and data reside (including loginout, command language interpretation, and image activation). The VMS executive server (seen in Figure 1 with diagonal shading) has special responsibilities that include bootstrap of the VMS environment. Note that hardware reboot is not required to restart VMS-on-Mach. The executive server is also responsible for managing server registration and access, as well as for managing the complete set of VMS user clients (processes).

2.2.2 The Process in the VMS Model

In order to preserve the behavior of the VMS process in our model, the user client evolved into a pair of closely-coupled Mach tasks. The particular behavior we wanted to preserve included:

- separation of the process itself from images executed within the process (in particular, protection of the process from erroneous user images),

  and

- protection of certain code and data from access by users.

One task, the user space task, provides the address space within which VMS user images are executed. Calls to the VMS application programming interface (API) by an image are acted upon by a read-only user library that resides in each user space task. The user library implements entry points for VMS system services in what is known as the VMS transfer vector. These entry points redirect system service calls to the routines which implement the services. Since the transfer vector is at a fixed location in the user image's address space for all VMS systems, programs linked against the transfer vector are assured that they need not be relinked for subsequent versions of VMS. This guarantee can be made because changes internal to VMS are hidden from applications by simply changing the contents of the transfer vector. In our model, system service calls are translated by the user library into service requests that may be handled directly within the user library, or directed to the appropriate VMS server task. All of this activity is invisible to the user image, allowing some freedom for the underlying operating system environment to evolve and change.

The user space task is created and managed by a second task, the process server. This server contains code and data that are protected from user access. The process server is designed to accommodate other operating system personality environments in addition to VMS. Code and data that are generic with respect to the operating system personality environment, like system access validation, receipt of exceptional conditions, and communication with other servers, are separate from and used to invoke operating system personality features, like command language interpretation, system services, image services, and exception processing. In our model, this personality-dependent piece is dynamically loaded during process server initialization.

This model of the VMS process also affected how the state of a process (its process context) is maintained. A main VMS software structure, the process control block (PCB), contains information managed by many VMS subsystems. With our model partitioning VMS into a set of servers, we decided to distribute the PCB information among the servers within the VMS environment. This favors performance of VMS servers in responding to user requests over performance of process-related information-gathering applications.

2.2.3 Servers in the VMS Model

Our model is based on a client-server approach. Individual subsystems within VMS are implemented as multi-threaded Mach server tasks. The functions of each VMS server are kept simple to provide basic building blocks from which the VMS personality environment may be constructed. This approach helped us focus on our goals of architectural independence and subsystem partitioning. With these goals, we see benefits to VMS that include:

- the ability to take advantage of widely-available and quickly-evolving hardware platforms with minimal effort and in a timely manner,
- improved reliability and maintainability,

  and
- increased flexibility to configure and evolve the system.

In order to provide some common framework for servers, as well as to isolate servers from the underlying kernel environment, we designed a server library. This library provides services that include basic task and thread management, synchronization mechanisms like mutexes and condition variables, error handling, inter-task communication, and memory management.

Although servers perform vastly different functions, their structure exhibits a number of similarities. Each server maintains static resources (typically data structures) for each client (typically a VMS user client). Servers provide communication ports to users and other servers. One or more threads in each server process requests sent to these ports.

Our model contains a number of servers that support the VMS environment. The remainder of this section discusses several of the basic servers and mentions a number of others servers included in the model.

Most functions performed by the VMS memory manager are replaced by equivalent functions within the

Mach kernel. Two memory management servers are included in our model to provide additional VMS memory management features: a file-mapped memory manager for implementing virtual address space mapped to files, and a shared memory server to implement shared address space mappings. The shared memory server relies on either the default Mach memory manager (pager) or the file-mapped memory server to manage backing store.

In our model, the Mach kernel creates a task for the VMS executive server. The executive server is the boot server that brings up the rest of the VMS personality environment. Once it has initialized, each VMS server registers its communication ports with the executive server. The executive server maintains these communication ports and makes them available to requesting servers. During normal system operation, the executive server creates VMS user clients, globally manages these processes, responds to queries for process information, and eventually deletes the user clients. In doing so, the executive server maintains the VMS notion of jobs.

The process server that manages each VMS user client provides three categories of services: system interface services, API services, and process state management services.

- System interface services include process initialization, system access validation, loading and presentation of a selected operating system interface (command language interpreter or shell and image services), and process termination.
- API services respond to requests from the user library on behalf of user images.
- Process state management includes services that maintain process attributes such as privileges and resource limits, execution state, exceptional conditions, and user space task control.

Other VMS features or subsystems that we modeled as servers include: accounting, auditing, authorization, common event flags, device drivers, error logging, the file system, I/O services, license management, lock management, logical naming, mailboxes, queue management, record management, the security database, and transaction processing services. This is not a complete list, but is indicative of the approach we used to partition VMS.

2.2.4 Internal Mechanisms

This section explores how VMS internal mechanisms, some of which are visible to users, are handled in our model. Certain mechanisms are no longer used within our model but are preserved for use by VMS applications (sometimes with changes that may be visible to applications).

2.2.4.1 Agendas and Notices

User-invoked system services often see delays in the processing of requests and usually have special semantics associated with the completion of these requests. In order to implement such user-requested VMS actions in an architecture-independent manner and to keep VMS server interfaces simple, we use a mechanism consisting of agendas and notices. Agendas and notices provide a single mechanism for storing and processing the semantics of actions to be taken when a service request completes. These actions, like setting an event flag upon completion of an I/O operation, remain local to user clients. An agenda structure is created by the client to remember the completion semantics associated with an operation. A unique identifier associated with this agenda structure is passed as part of the server request. When the server completes the requested operation, a notice message including the agenda structure identifier and any request-specific status or data is sent back to the client for processing. Pre-defined agendas and notice messages are also used to handle unsolicited operations, like a hardware exception.

2.2.4.2 Asynchronous System Traps (ASTs)

An AST is a mechanism that enables an event to alter the flow of control in a VMS process. Users may request AST notification when using VMS system services, and VMS itself requests ASTs as a result of some operations. Our model replaces the use of ASTs within VMS by notices and agendas, while maintaining the AST mechanism for user images.

When a user image requests a service that will eventually queue an AST, an agenda structure containing the AST function address and parameter is created. When the service request has completed, a notice message is issued back to the user space task. A thread within the user library suspends the user image thread, redirects it (by pushing a call frame on the stack) to an AST dispatch routine, and then resumes the user image thread. This dispatch routine calls the user image's AST routine with its associated AST parameter. When the user's AST routine completes, a message is sent back to the user library thread indicating AST completion. AST completion indicates that subsequent ASTs can now be delivered. (VMS dictates that only a single user AST may be active at any one time.) Note that the chain of stack frames is not guaranteed to be the same as in the current VMS implementation.

VMS servers use agendas and notices to mimic the function of ASTs. Servers maintain agenda structures to keep track of asynchronous operations currently in progress. Upon receiving a notice message, a thread within the server executes the functions specified by the agenda.

We also looked into the possibility of problems that could result from deviating from the existing VMS AST implementation. One of these deviations is the lack of access mode prioritization of ASTs and another is execution of code (usually privileged) in user address space. Our investigations uncovered only a few cases that needed to be addressed, all of which could be handled using Mach-provided functions.

2.2.4.3 Exception and Condition Handling

In the Mach exception handling facility [3] there is machine-dependent and machine-independent kernel software to deal with exceptions. Machine-dependent software implements exception dispatch and servicing that is specific to each hardware platform. The exception servicing code packages the information for processing by machine-independent software. This processing often consists of sending the exception out of the kernel for handling and then following up on the results of that handling. This model works well for handling VMS exceptions and conditions. When a user image exception occurs, a remote procedure call (RPC) message is sent to the process state management port within the appropriate process server. Processing of a user image exception causes the user image thread to be suspended and an IPC to be sent to the user library thread to initiate a search for a condition handler. While the condition handler search proceeds in the context of the user space task through user library thread actions, the process server returns from the RPC. Note that the chain of stack frames is not guaranteed to be the same as in the current VMS implementation. If the search is unsuccessful, or if all handlers resignal, the user library thread issues an IPC back to the process server indicating that the condition was not handled successfully.

2.2.4.4 Interrupts, IPLs, and Spinlocks

In VMS, interrupts (like exceptions) are events that require the execution of software other than the current thread of execution. Unlike exceptions, however, interrupts are unrelated to the current thread of execution and are asynchronous to it. To arbitrate among interrupt requests, each request has an associated interrupt priority level (IPL). When an interrupt is granted, processor IPL is raised to that of the interrupt request and the interrupt is handled by a service routine. Interrupt requests with an IPL that is the same or lower than the current processor IPL are blocked.

IPL applies separately to each processor in a multi-processor configuration. When symmetric multiprocessing (SMP) was implemented in VMS, spinlocks were introduced to synchronize access to shared kernel resources. To prevent deadlock, spinlocks are ranked according to their associated IPL. A thread may not acquire a spinlock with an equal or lower ranking than any spinlock currently held by the thread.

In our model, the server library provides low-level primitives for manipulating threads of control. These primitives include forking and joining of threads, protection of critical regions using mutex variables, and synchronization by means of condition variables. Spinlocks are replaced by mutexes that are similarly ranked. VMS server threads may acquire any mutex desired, but all mutex acquisition must be done in order of increasing rank. To acquire a mutex of lower rank, a server thread must first release all higher ranking mutexes. The scheduling effects of VMS spinlocks are not maintained by VMS server mutexes. Although we have not investigated these effects, a similar change occurred when BSD UNIX was trans-

formed into a single server on Mach and no noticeable effects were observed. In summary, we rely on a server mutex acquisition protocol to achieve the required synchronization.

Hardware interrupts related to I/O devices are caught and serviced in the machine-dependent code of the Mach kernel. This is similar to interrupt-handling code currently in VMS. Where in the current VMS implementation the interrupt handler causes further processing by posting a lower-level interrupt to awaken a VMS fork process, in our model further processing is caused by the interrupt handler sending a message to the appropriate device driver server.

VMS threads of execution that are started as a result of software interrupts do not exist in our model. This activity is mimicked by VMS server threads and server library functions in the model.

3 VMS Prototype Implementation

The VMS-on-Mach prototype is a subset implementation of our model. During its development many design choices were made to facilitate its implementation. At present, our prototype is capable of supporting multiple VMS processes and running several existing VMS images. In addition to the many test programs we wrote during its development, the prototype successfully executes the following standard VMS images without modification:

- CREATE, used to create a text file,
- COPY, to make a copy of a file,
- TYPE, to list the contents of a file,
- SEARCH, to search for a string within a file,
- DIFF, to compare two files and report differences,
  and
- EDT, to perform line-mode editing on a file.

Figure 2 at the end of this section illustrates a sample session using the prototype, the development of which is discussed in greater detail in the following sections.

3.1 Approach and Objectives

Early in the project we decided that we would use a VAX platform to produce the prototype. We chose the VAXstation 3100-48 system after discussions with the CMU Mach team, who provided us with a version of Mach 3.0 on that system. We decided that this choice would best help us in demonstrating our model as it would allow for the execution of existing VAX/VMS non-privileged (user) binary images. We considered using a MIPS-based DECstation to demonstrate architectural independence for our prototype. This approach was rejected, however, as it would not have allowed us to use existing VMS sources or images.

We also wanted to take advantage of existing VMS and UNIX tools as much as possible, rather than create a new operating system and software development environment from scratch. This resulted in a VMS-on-Mach prototype that incorporates a set of servers into the existing Mach 3.0 based 4.3 BSD UNIX-compatible single-server system [6]. We built VMS servers as separate Mach UNIX processes, allowing us to take advantage of Mach threads and the BSD server. These VMS servers are for the most part new C code. Our decision to rewrite versus rework existing code revolved around how expedient it was to understand and rewrite the code, while considering how much effort was required to utilize existing sources. For most of the prototype work done to-date, rewriting was more expedient. This may or may not be true in a more complete prototype that includes, for example, the VMS record management and file systems. Our prototype implements only a small subset of the existing RMS interface on top of the UNIX file system (UFS).

Our objective for the prototype stage of the project was to implement a subset VMS environment that:

- involves multiple server tasks,
- isolates platform dependencies,

  and
- allows for some existing VMS non-privileged (user) binary images to be executed.

There were two phases to the prototype effort: a framework building phase, during which the infrastructure needed to execute simple VMS images was created, and an environment building phase, which added VMS services needed to execute more complex images. These phases are described in the following sections.

3.2 Framework Building Phase

Our goal for this phase, which lasted about three months, was to execute a simple VMS image. Work during this phase of the prototype effort concentrated on setting up the development environment, creating a server library, and building an implementation of the process server.

Our development environment is structured as a set of hierarchical directories which reflect the organization of the model. This structure is deliberate and designed to prevent unexpected cross-dependencies between the various components of the prototype. Because our software is designed using a client-server approach, there are many cross-component dependencies. This led us to develop a three-phase software build process. The first phase, interface export, descends the hierarchy generating and exporting public interface files (*e.g.*, generated IPC/RPC definitions and routines, and server-specific service definition files). The second phase, object library generation, also descends the hierarchy, this time building object libraries and exporting these libraries up the hierarchy. The final phase, image generation, creates the various servers by linking the required libraries for each image.

During early design discussions we investigated an optimization that would combine or bundle servers together into a single image. This optimization would allow for easier development and testing while providing optimized performance. These discussions evolved into a design philosophy that allows (in fact, requires) servers to be developed and tested independently of other servers, but permits these servers to be bundled into a single image at image generation time. This design philosophy is incorporated into our vbundling tool. The vbundling tool reads a file that describes all of the servers in the prototype and all of the server images that are to be generated. This tool allows us to alter the bundling configuration of the prototype by modifying a single file (the vbundling script) and rebuilding the prototype. Note that a change in the vbundling description file results in only the relinking of server images.

The first component of the prototype to be developed was the server library. Although our prototype was developed on Mach 3.0 using Mach UNIX facilities, we decided to hide as many of the underlying kernel and UNIX services as possible. This means that, in general, any Mach or UNIX service used by a server is provided by the server library. In some cases, however, it was not practical to build a server library interface to a Mach or UNIX service due to its use in a server as an expediency. For example, the authorization server uses the getpwent() UNIX service directly to scan the /etc/passwd file.

Initially, we defined three areas that the server library needed to address: memory management, server threads, and kernel task and thread management. Since each server is a UNIX process, the memory management services are implemented using the malloc() and free() functions. Server threads (sthreads) are a mechanism that allows servers to create multiple logical threads that either share or map directly to kernel threads. Since sthreads are close in function to the cthreads package [10] provided with Mach, we chose to implement them as a layer on top of cthreads. Since our design has VMS images executing within the context of pure Mach tasks (*i.e.*, not UNIX processes), we provided services in the server library for creating and managing Mach tasks and threads. To support architectural independence, we added functions for managing user image call frames that are used to build initial as well as interrupt call frames (*e.g.*, VMS ASTs or UNIX signals).

Our model defines several generic services that are used in the management of VMS processes (*e.g.*, process creation and termination). These services are defined as subsystems using the Mach Interface Generator (MIG) [10], which creates client-side and server-side bindings for each service as well as a subsystem dispatch routine. Subsystem dispatch routines are responsible for demultiplexing the set of IPC/RPC messages used by the subsystem into calls upon the appropriate server routines. Due to our server bundling methodology, each server's server-side binding for a particular routine must be uniquely named in order to avoid name conflicts when bundling several servers into a single image. As a result, we needed to define services that had only a single client-side binding but multiple server-side bindings. Our design also allows IPC/RPC service messages defined by different subsystems to be received and processed on a common port. Because of this, we could not use the MIG-provided library function to receive and dispatch messages to their service routines. To solve this problem, we implemented a server library object, the demux, which allows several subsystem dispatch routines to be grouped together. We then added a server library function that receives and processes the messages using the dispatch routines associated with a demux. This function optionally creates new threads for the processing of each received message.

Support of VMS image activation was initially developed in C on a VMS system, where we were able to verify the actions of our implementation with information obtained using the VMS ANALYZE/IMAGE program. Once the image activation code was debugged, we moved it over to our prototype. As an optimization, we load and prepare the user image within the process server's address space, and make use of Mach memory inheritance to set up the user image address space. This presented us with two problems: conflicts in the base addresses of the process server and user image, and conflicts in the location of the VMS transfer vector and the Mach UNIX emulator. VMS images have a default base address that overlaps addresses in use by the process server. As mentioned earlier, VMS uses a transfer vector to process system services. The location of the VMS transfer vector overlaps the Mach UNIX emulator that is part of every Mach UNIX process. We solved these problems by using existing VMS tools. VMS compilers generate position-independent code. This made it possible for us to use a VMS linker option to alter the base address of the image (thereby moving it out of the way of the process server). We also created an alternate version of the system service transfer vector at a different address. This allowed us to generate images that do not overlap the process server or the Mach UNIX emulator, and that are essentially the original user images. These changes allow us to load and prepare VMS images within the process server's address space.

Our next step was to build a framework for running VMS images. Our first user library function was implemented to support the starting of a VMS image. This function is called with the image's starting address information and is responsible for calling the image's main routine. Image completion, including any final status value, is signaled to the process server by the user library when the image's main routine returns or the $EXIT system service is invoked. Our initial VMS images were run by creating an sthread within the process server that executed the user library start function. Our next step was to logically separate this mechanism from the process server by using IPC for synchronization. Finally, we created a Mach task and thread to run the VMS image in a separate address space. This last step (separate tasks) was particularly difficult as we did not have a debugger that operated on pure Mach tasks.

3.3 Environment Building Phase

Our goals for this phase, which lasted about three months, were to expand the prototype and demonstrate a subset VMS environment that allowed users to login and logout, as well as execute some VMS programs. These programs included variations on *hello world* and existing VMS utilities.

The Digital Command Language (DCL) is the primary command language for VMS. DCL differs from other command languages in that the syntax for all commands is described in a set of files that are compiled into the special VMS image known as DCLTABLES. Each DCL command either invokes an internal DCL function or executes a specific VMS program. As we did with image support, we developed C code on VMS that read and parsed DCL commands. When our code was ready, we moved it to the prototype and integrated it into the process server. This required modifications to the existing image services to support reading the DCLTABLES image.

We also developed a simple authorization server that validates usernames and username/password combinations against the UNIX /etc/passwd authorization file. The process server was modified to request login information and validate the login request by using services provided by the authorization server.

At this point of the prototype we had two servers: the process server and the authorization server. Communication ports for these servers were managed by the system name server (snames) provided with Mach UNIX. Recognizing that we would soon start building additional servers, we developed a set of port name services within the server library to provide port registration and look-up functions. These services, which were initially developed as an interface to the system name server, were later modified to use the executive server as the prototype's port name server.

The executive server was developed with the primary intention of managing servers and processes in the prototype. Initially, the executive server acted only as a name server which processed server and process registration requests and converted port name look-up requests into authorized ports. As the prototype evolved, the executive server was enhanced to support system and process information queries, and to read a configuration file containing start-up information. Port name services are provided to other servers by the server library (which handles communication with the executive server). Information needed to process system queries is defined in the executive server's configuration file, which also contains a list of server images that should be automatically started by the executive server. The executive server uses the UNIX fork() and exec() functions to start other servers. The executive server supports process information queries by forwarding these queries to the appropriate process server. This implementation allows cross-process queries to be directed to individual process servers in the prototype.

As the number of servers increased, we found it increasingly difficult to debug the prototype. Consequently, we added a debug facility in the process server. Ideally the debug facility should be associated with the executive server or system console, but since the process server was the only server that performed terminal input, it was the obvious choice. Each server was modified to contain an array of debug flags that could be enabled or disabled via special IPC calls. A command line interface was built that allows any process server to communicate with a specific server and enable or disable any combination of debug switches. These on-the-fly debug switches have proven quite valuable in debugging the prototype.

File and terminal services are provided by the I/O server. Given our time and resource constraints, we first identified the minimum set of services we needed. The I/O server provides disk I/O that is used by image activation, the file-mapped memory server, and all file and data operations required by the VMS record management (RMS) services implemented in the user library (open, close, read, write, and flush). It additionally performs all terminal input and output for the prototype. The I/O server is implemented using the UNIX file system services provided in the Mach UNIX environment. Its public interfaces, however, provide services that are more generic than the existing UNIX interfaces, and are used to facilitate the implementation of RMS in the user library. All reads and writes are currently limited to a maximum of 512 bytes per operation and are implemented using the UNIX read() and write() functions (*i.e.*, they are treated as raw reads and writes).

VMS uses the notion of section files to refer to disk files that are mapped into memory. The most common use of section files by VMS is image activation. Each VMS image contains multiple image sections, each of which is a series of 512-byte blocks that are mapped to various locations in memory. The file-mapped memory server implements section files for the prototype. The process server communicates with the file-mapped memory server to initialize mappings for various portions of image files. The file-mapped memory server creates memory objects to represent these mappings, and is the Mach external pager for the objects.

As our prototype development continued, we were able to optimize many user services by moving them from the process server into the user library (*e.g.*, condition handler search and AST processing). This led us to revise our model and design of the user library. In the new design, the user library contains a thread which acts as the first-line manager for the user image. This thread receives and processes completion notices from other servers and initiates condition handler searches and AST delivery.

When the user address space is created to run a user image, the user library thread is started. This thread initializes the user library and informs the process server when initialization is complete by returning a port which the process server can use to communicate with the user library thread. Upon receipt of this user library port, the process server starts the user image thread, which is responsible for calling the main routine of the image. When the user image exits, an image completion message is sent to the process server, notifying it to terminate the user space task and resume processing in the command language interpreter (CLI). Note that if either the user image thread or user library thread incurs an exception that is not successfully handled, the process server terminates the user space task and resumes processing in the CLI.

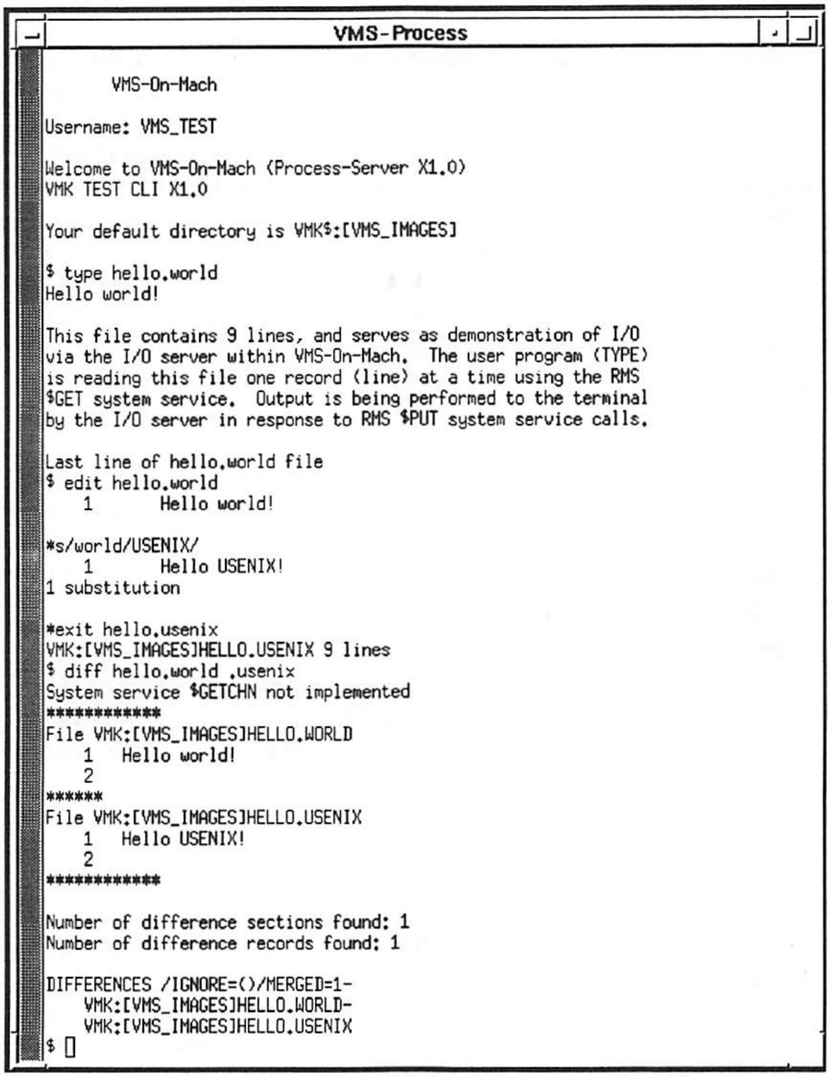
VMS uses logical names quite extensively. Logical names provide a mapping of a string to zero or more replacement strings. They are grouped into tables which are either private to a process or shared within a process tree, a login group, or across the system. Note that VMS provides extensive mechanisms for protecting and authorizing access to shared logical name tables. A simplified logical name server was implemented in the prototype. This server implemented a single logical name table that is unprotected and is shared among all servers and processes. The logical name server supports a configuration file that allows for the definition of names. RPC services are provided for requesting logical name translations. These services are used by the process server and user library to resolve VMS file names.

In summary, our prototype implements the following components:

- a user library that implements the VMS transfer vector and system services,
- a server library that isolates kernel primitives from the servers,
- an executive server that provides server and process registration and look-up services,
- a process server that implements DCL and manages the user space task,
- a file-mapped memory server that provides support for section files,
- an I/O server that provides access to terminals and files,
- an authorization server that validates login requests,
- a logical name server that provides name translation services,

  and
- a common event flag server that provides multi-process synchronization services.

Figure 2 on the next page shows a sample prototype session that illustrates the use of the process server, the executive server, the authorization server, the I/O server, the file-mapped memory server, and the logical name server.



```
VMS-Process

      VMS-On-Mach

Username: VMS_TEST

Welcome to VMS-On-Mach (Process-Server X1.0)
VMK TEST CLI X1.0

Your default directory is VMK$:[VMS_IMAGES]

$ type hello.world
Hello world!

This file contains 9 lines, and serves as demonstration of I/O
via the I/O server within VMS-On-Mach.  The user program (TYPE)
is reading this file one record (line) at a time using the RMS
$GET system service.  Output is being performed to the terminal
by the I/O server in response to RMS $PUT system service calls.

Last line of hello.world file
$ edit hello.world
    1       Hello world!

*s/world/USENIX/
    1       Hello USENIX!
1 substitution

*exit hello.usenix
VMK:[VMS_IMAGES]HELLO.USENIX 9 lines
$ diff hello.world .usenix
System service $GETCHN not implemented
************
File VMK:[VMS_IMAGES]HELLO.WORLD
    1   Hello world!
    2
******
File VMK:[VMS_IMAGES]HELLO.USENIX
    1   Hello USENIX!
    2
************

Number of difference sections found: 1
Number of difference records found: 1

DIFFERENCES /IGNORE=()/MERGED=1-
    VMK:[VMS_IMAGES]HELLO.WORLD-
    VMK:[VMS_IMAGES]HELLO.USENIX
$
```

Figure 2 VMS-on-Mach Prototype Session

4 Conclusions

Developing the VMS model and implementing the prototype software was a valuable experience for us. In this section we present lessons we learned and future directions we believe useful to pursue.

4.1 Findings

We found the Mach abstractions helpful both in designing the model and in implementing the prototype. The client-server approach that Mach supports was very useful conceptually in designing our partitioned model of VMS-on-Mach. In implementing the prototype we also found that the partitioning lent itself to ease of construction and debug. However, we also found issues for Mach. This section describes these findings.

4.1.1 Authentication

We found it efficient to have VMS servers conduct authentication once for each client, and on success reward the client with the assignment of a server port. Since Mach tasks cannot manipulate ports except via Mach kernel calls, we can be sure that all messages delivered through this port originate from the client to which the port was assigned. Furthermore, Mach developers pointed out that the port name can be set to a virtual memory address, and, since this name is supplied by the Mach kernel for received messages and cannot be forged, it can be used to point directly to the data structure element corresponding to the client to which the port was assigned. This eliminates a number of checks that would otherwise be required. Use of ports in this way means that for each class of service, where we would otherwise use a single service port, we now use one port for each client so that we can identify the client using the port. The total number of ports required is therefore on the order of the number of service classes multiplied by the number of clients. The resource implications of this require further study.

4.1.2 VAX Access Modes

The VAX architecture defines four access modes, which implement a protection-ring scheme for limiting access to data residing in the virtual address space. In place of rings superimposed on a common virtual address space, our model uses separate virtual address spaces in the form of separate Mach tasks, resulting in a more constrained strategy for data sharing.

The VAX architecture allows the current VMS implementation to access data of less privileged modes using direct memory references. With a client-server model, access to this type of data typically requires one or two IPC messages. Our model minimizes the need for these IPC messages by data partitioning and localized access. We did not, however, encounter any difficulties in preventing access to data of more privileged access modes.

4.1.3 Scheduling and Memory Management Policies

It is not feasible to reproduce the VMS scheduling policies using Mach. VMS decrements the size of the time quantum assigned to the currently executing thread each time the time quantum expires, and boosts the priority of the currently executing thread when involuntary waits occur. While the exact VMS semantics may not be possible, the Mach time sharing scheduling policy [2] does have the same basic goals as the VMS scheduling policy, namely to be responsive to interactive processing while preventing starvation. Therefore the difficulty in implementing the exact semantics of the VMS algorithm at the server level is not a major issue. An exception to this occurs in the case of deadline scheduling, which may be needed for implementing network protocols.

A similar situation exists with regard to memory management. Mach does not use the same mechanisms as VMS to balance free-page deficits. Swapping, for example, is not used by Mach. We also did not attempt to model VMS process working sets (the process' virtual pages that are currently valid and in physical memory). The Mach kernel does not currently export an association of resident pages to tasks or allow externally influenced per-task page replacement, both of which are needed to implement working sets.

4.1.4 Kernel Resource Management

Management of kernel resources is needed to isolate the ill effects of any one VMS user image from other VMS processes and servers, and indeed from future user images of the same VMS process. Tasks that do not cooperate must not be able to interfere with each other by tying up resources. Such management is performed within VMS by the use of quotas. The Mach kernel does not supply any way to impose limits, such as quotas, on tasks. In an implementation of a production operating system environment, a resource model is needed to control memory and processor usage on a task basis.

4.1.5 Device Drivers

In the version of the Mach kernel used in our model, device drivers reside in the kernel. While an interrupt service routine needs to be located in the kernel, it should provide minimal function, relegating most of the work to a user-level task. Also, device drivers can be partitioned to reduce the amount of replicated code. These modifications are being addressed by Mach developers at CMU [4].

4.2 Future Directions

We have not fully addressed a number of topics that we nevertheless feel are important to explore if VMS-on-Mach is to become a practical operating system environment. These topics are described in this section.

4.2.1 Performance Optimizations

In order for VMS-on-Mach to be a viable alternative, it must have performance comparable to that of VAX/VMS. Preliminary measurements indicate that unoptimized performance is likely to be unsatisfactory, but large gains can probably be made through judicious optimizations that do not detract from the advantages of the multi-server model. In order to identify the needed optimizations, we must carry out carefully designed measurements. Most of these measurements (and therefore most of the resulting optimizations) can only be made after more of the VMS-on-Mach model has been implemented. Our thoughts on possible optimizations are described here.

4.2.1.1 Proxies

A proxy [10] is a section of code and/or data resident in a client's address space, which acts as an agent for a server. The client may be a user space task or a VMS server. The proxy is considered part of the server and can be dynamically loaded into the client by the server. The advantage of using a proxy is that a client's address space can be accessed directly. Invoking a proxy incurs only the cost of a local procedure call instead of the cost of an RPC to the server.

One of the primary uses of a proxy is to provide a client-side access window to information maintained by a server. Proxy use can considerably reduce client-server communication, thereby reducing the overhead and latency of accessing the data.

Currently we have not identified any specific use of proxies, but we believe the concept would be useful in a full operating system implementation using our approach. For instance, one possible use might be in implementing I/O read buffering.

4.2.1.2 Distributed Data

Strictly speaking, the client-server paradigm used in our model dictates that a single server is responsible for managing a specific piece of data, and that all references to the data are performed by RPCs to the managing server. This philosophy makes updates cheap but references expensive, and is inefficient for data that is frequently referenced but seldom modified. For this reason, we envision relaxing this paradigm to allow copies of selected data to reside in multiple servers, with an RPC being used to update these copies. To date, however, we have not identified specific uses of this technique.

4.2.1.3 Message Transport Independent of User Interface

The Mach kernel provides in-line and out-of-line modes of message transport. With in-line mode, the data is sent along with the message; with out-of-line mode, the message specifies the data location but does not actually include the data. In-line mode is appropriate for short messages while out-of-line mode is preferable for long messages. The choice of passing data in-line or out-of-line is an optimization that should be made by the message subsystem. Software changes should not be required.

If two servers that send messages to each other are bundled into the same task, server code should not be required to optimize the transport mode used to send messages between these servers. For instance, shared memory might be utilized as the basis for an optimized transport mode.

To avoid intrusion of optimization decisions into the source code of VMS-on-Mach, we looked into designing a mechanism in our model so that the message transport mode may be chosen automatically, and the user does not need to explicitly indicate the transport mode.

4.2.2 Other Interesting Topics

A number of important operating system features and subsystems were not examined in our effort so far. These need to be incorporated into our model.

VAXcluster technology provides users with an available and distributed computing environment. Multiple computing resources are used to create a cohesive environment that facilitates the dynamic sharing of devices and files. The use of micro-kernel and client-server technologies provide us with additional opportunities for delivering distributed computing. In light of this we need to examine how these technologies might be used to provide for, and enhance, the features of VAXcluster systems.

We need to define the generic I/O, file system, and record management services that the I/O server, file system server, and record management server will provide, and identify the work that must be done in the user library to build an I/O subsystem for user images. High performance of the I/O and file system servers is crucial to the success of our model. This is achieved by ensuring that overhead-inducing layers of software can be bypassed if their capabilities are not required.

We are interested in exploring the issues related to orchestrating multiple operating system environments on the same kernel, as well as to cooperation among distributed (multi-) computing systems. For example, a philosophy is needed to provide for management of underlying kernel resources and multiple personality environments. In addition, thought needs to be given to the networking model for such a system.

References

[1] Accetta, Mike, Baron, Robert, Golub, David, Rashid, Richard, Tevanian, Avadis, and Young, Michael, *Mach: A New Kernel Foundation For UNIX Development*, Technical Report, School of Computer Science, Carnegie Mellon University, August, 1986.

[2] Black, David L., "Scheduling Support for Concurrency and Parallelism in the Mach Operating System", *Computer*, vol. 23, no. 5, pp. 35-43, May, 1990.

[3] Black, David L., Golub, David B., Hauth, Karl, Tevanian, Avadis, and Sanzi, Richard, *The Mach Exception Handling Facility*, Technical Report CMU-CS-88-129, School of Computer Science, Carnegie Mellon University, April, 1988.

[4] Forin, Alessandro, Golub, David, and Bershad, Brian, "An I/O System for Mach 3.0", *Proceedings of the USENIX Mach Symposium*, pp. 163-176, November, 1991.

[5] Gien, Michel, "Micro-kernel Design", *UNIX REVIEW*, vol. 8, no. 11, pp. 58-63, November, 1990.

[6] Golub, David, Dean, Randall, Forin, Alessandro, and Rashid, Richard, "UNIX as an Application Program", *USENIX Summer Conference Proceedings*, pp. 87-95, June, 1990.

[7] Goldenberg, Ruth E. and Kenah, Lawrence J., *VAX/VMS Internals and Data Structures, Version 5.2.* Digital Equipment Corporation, Order No. EY-C171E-DP, 1991.

[8] *IEEE Standard Portable Operating System Interface for Computer Environments, IEEE Std. 1003.1-1988*, Institute of Electrical and Electronics Engineers, Inc., 1988.

[9] Loepere, Keith, *MACH 3 Kernel Principles*, Open Software Foundation and Carnegie Mellon University, 1991.

[10] Loepere, Keith, editor, *Server Writer's Guide*, Open Software Foundation and Carnegie Mellon University, 1990.

[11] Rashid, Richard, "A Catalyst for Open Systems", *Datamation*, vol. 35, pp. 32-33, May 15, 1989.

The Increasing Irrelevance of IPC Performance for Microkernel-Based Operating Systems

Brian N. Bershad
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Brian.Bershad@cs.cmu.edu

March 10, 1992

Abstract

IPC is the glue with which traditional operating system services such as networking, and filing, are provided in microkernel-based operating systems. Because applications rely heavily on cross-address space communication, IPC performance is often viewed as being the "Achilles heel" of a microkernel-based operating system. In this paper I discuss four reasons why IPC performance is becoming increasingly irrelevant to overall system performance.

1 Introduction

Microkernels such as V [Cheriton 84], Chorus [Rozier et al. 88], and Mach [Accetta et al. 86] provide the infrastructure with which other operating systems such as Unix, MS-DOS and VMS can be implemented as user-level programs [Golub et al. 90, Rashid et al. 91, Cheriton et al. 90, Wiecek 92]. Because applications use cross-address space IPC to interact with traditional operating system services, IPC performance has been thought to be the Achilles heel of microkernel-based systems. Even with the improvements in IPC performance that have occurred in the past 5 years [Bershad et al. 90, Draves et al. 91], microkernel based systems are still believed to have *inherently* worse performance than their monolithic counterparts. This belief stems from the fact that the system call, which is the mechanism for interacting with monolithic operating systems, such as

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035 and in part by the Open Software Foundation (OSF), and a National Science Foundation Presidential Young Investigator Award.

4.3BSD [Leffler et al. 89] and Sprite [Ousterhout et al. 88], is faster than a cross-address space IPC, which is how applications interact with user-level operating system services.

Although IPC latency in microkernels is slower than system call latency, the absolute difference between them has reached the point where it can be largely ignored. In other words, IPC performance is becoming increasingly irrelevant as a metric with which to assess microkernel viability.

There are four reasons that IPC performance should no longer be a principal metric by which one judges the "goodness" of a particular operating system microkernel, or even a particular approach to building operating system microkernels. In brief, these reasons are:

1. IPC has gotten faster faster than the rest of the operating system.
2. Performance is dominated by caches, not by address spaces.
3. All data does not need to be marshalled through the kernel.
4. All services do not need a hardware firewall.

The first two reasons stem from the ever-growing performance imbalances which exist in today's systems. Simply stated, IPC mechanisms are not the performance bottlenecks which they once were. Instead, there are other, more fundamental bottlenecks, such as memory transfer speed, network latencies, disk speed, and cache management overhead. The second two reasons are due to the maturity which microkernel-based systems have achieved in the last few years. Specifically, in their efforts to increase performance, systems builders have discovered a collection of techniques which can be used to bypass, or at least "dance around" IPC facilities.

In the rest of this paper I expand on the reasons listed above, and discuss why we should stop measuring microkernel systems by the speed of their round trip IPC times. I present examples and observations from the Mach 3.0 microkernel running on a collection of architectural platforms to motivate and substantiate the discussion.

2 IPC Has Gotten Faster Faster Than the Rest of The Operating System

It has been observed that operating system performance has improved far less rapidly than would be expected given improvements in processor architecture and implementation [Ousterhout 90, Anderson et al. 91]. Although the time to add two registers together has increased by almost two orders of magnitude in the last decade, the performance of key operating system services, such as filing, paging, and networking, has remained relatively flat. Disks continue to spin at about the same speed as they always have, buffer caches remain limited by memory bandwidth, and core network latency remains on the order of several hundred microseconds (although network bandwidth has improved somewhat more dramatically).

In contrast to the performance of the services which are being provided by the operating system, the time to send a message between two address spaces on the same machine has dropped substantially. The reasons for this are two-fold. First, we've become more careful and more successful at building IPC mechanisms "for speed," ruthlessly streamlining and optimizing the common cases. Using the Mach 3.0 microkernel as an example, IPC performance on a Microvax-III (CVax processor) has gone from about 750 μsecs for a round-trip RPC in 1989 [Bershad 90] to 497 μsecs in 1992 (measured recently using Mach 3.0 version MK68). The improvements were due to tightening the interface [Draves 90], and the implementation [Draves et al. 91].

The second reason why IPC has gotten faster faster than the rest of the operating system is that measured IPC performance has, at least to this point, tracked processor

performance reasonably well.[1] In Mach, this is largely because the improvements made in the last several years have moved IPC performance off of the memory curve and onto the processor curve by tightening the locality of the IPC paths. Again, using Mach 3.0 MK68, a round-trip RPC takes 57 μsecs[2] on a DecStation 5000/200. That system, which uses a MIPS R3000 processor running at 25Mhz, is rated at roughly ten times the performance of the CVax. In keeping with this, cross-address space RPC is about nine times faster than the same code running on a CVax.

With IPC performance improving more rapidly than general operating system service performance, the "hit" required to access such a service is becoming less important. For example, on the older CVax, the 497 μsecs required to do a cross-address space RPC was comparable to the time it took seek one disk track, or copy a 512 byte block from a system buffer cache into a user buffer, or to transmit a packet over an Ethernet. On the DecStation 5000/200, however, the IPC penalty is substantially less. Since disk access, data transfer, and network latencies have not improved at anywhere the same rate, the additional cost of the IPC required to indirect to these facilities has become less significant.

2.1 System Calls Are Not The Solution

System calls used in monolithic kernels are the alternative to the IPC mechanisms used in microkernels. Monolithic operating system services reside in the kernel, and are accessed by applications with a single kernel boundary crossing. On older systems, such as the CVax, the time to execute a Mach system call[3] was about 60 μsecs, substantially less than the IPC overhead (497 μsecs) on that machine. In contrast, on the newer DecStation 5000/200, system call overhead is about 8 μsecs, or about 50 μsecs less than a round-trip RPC. While the relative cost of IPC and system calls on the two machines is the same, the absolute difference *compared* to the service access cost (disk, buffer, or network), is much smaller. As a result, there is diminishing incentive to use system calls, rather than generalized IPC facilities, to access system services.

3 Performance is Dominated By Caches, Not by Address Spaces

The invocation of an operating system service implies a change in locality. In monolithic systems, the new locality is in the kernel. In microkernel-based systems, the new locality is in another address space. In both cases, however, the change in locality can result in an increased cache miss rate [Agarwal et al. 88, Mogul & Borg 91]. On older style, slower architectures (< 10 MIPS), cache miss penalties were only a few cycles. On today's architectures, however, cache miss penalties are tens, and soon to be hundreds, of cycles. These kinds of penalties can easily dwarf the kernel's IPC overhead. In effect, the cost of accessing an operating system service is going to be most influenced by whether or not that service is in the cache – not whether or not it's in the kernel or in another address space.

One could argue, however, that OS interaction via a microkernel actually involves two changes in locality – one to the microkernel and another to the server. While true,

[1] Although IPC performance is ultimately limited by architectural features such as trap and context switch time, and although these times have not been improving at the same rate as processor speed [Anderson et al. 91], practical IPC performance has not yet reached this limit.

[2] All times were measured with warm caches.

[3] Mach, although a microkernel, does export a small number of "true" system calls.

the microkernel's locality (at least on the critical path through to the operating system server) is small. The common-case round trip IPC path in Mach 3.0 on the MIPS, for example, requires less than 4KB of instructions and references less than 2KB of data, most of which is on the kernel stack. Because the locality is small, and identifiable *a priori*, one could decrease the chance that misses occur on the IPC path by allocating memory with an eye towards the cache hashing function [Bershad et al. 92]. For virtually addressed caches, this means devoting pieces of the machine's virtual address space to the microkernel. For a physically addressed cache, physical memory must be devoted, although one can play tricks with split instruction and data caches to ensure that only data pages conflict with designated instruction pages, and vice versa.

One place where cache performance becomes apparent is in the management of external devices which use DMA, and not programmed, I/O. Before the processor reads memory into which DMA has been performed, it must purge that memory from the cache to ensure that stale data is not returned. Before a processor issues a write request to a DMA device, it must flush the memory to be written from the cache, again to ensure that stale data is not read by the device. These cache operations are expensive. For example, on the HP-700, a high performance workstation based on the HPPA RISC processor with a cycle time of 20 nanoseconds, cache purge and flush operations take between 1 and 14 cycles per line (32 bytes), DMA operations tend to be page-oriented, so I/O operations require between 128 and 896 cycles simply to ensure memory coherency. In the case of device reads, performance degrades even further as the newly transferred data is faulted into the cache, at a cost of 16 cycles per line (2048 cycles per page). In contrast, a cross-address space RPC in Mach 3.0 on that machine takes about 70 μsecs (3600 cycles). Consequently, the *cost* of accessing an out-of-kernel device server (changing address spaces) represents only one component of the total CPU/device communication cost.

4 All Data Does Not Need to be Marshalled Through the Kernel

Microkernel-based operating systems can preallocate buffers between client and server address spaces. This allows an operating system service to share address space with applications, just as it did when the service was resident in the kernel. Small to medium amounts of data can be transferred from one address space to another by depositing it in the shared regions, rather than by sending it through the kernel in a message [Bershad et al. 91]. Large data segments (on the order of pages) can be passed using virtual memory primitives.

Co-mapping of IPC data has been used in several places. At CMU, applications share memory with the Unix server to pass data in and out of the file system. We have recently applied this technique to the socket interface as well. On a DS5000/200, we have found that the mapped socket interface does not improve the performance on small packets (fewer than 100 bytes), but in larger packets there is an improvement. For packets of 4KB, sends using the mapped socket interface are 15% faster than using the regular, non-mapped, interface. In packets of more than 4KB the mapped interface avoids the cost of dynamically allocating and deallocating the region for out-of-line transmission, although for extremely large packets remapping, rather than copying, is more efficient.

Co-mapping can be used effectively for non-Unix interfaces as well. For example, we have built a version of the X11 window server which uses Mach IPC for communication between X clients and the server. X does call request buffering, but clients frequently flush the buffer, which causes the data to be made available to the server. When Unix stream sockets are used to implement the transport layer, the flush causes a socket

write to occur. With sockets, multiple flushes may occur before the server collects any of the data, but all of the data can be collected in one receive, so transferring data through the kernel doesn't necessarily increase the number of context switches. When using Mach's IPC, which is message-oriented, every flush results in a message. Message boundaries are maintained, so every message must be collected separately by the X server, increasing the the number of user-kernel boundary crossings and context switches relative to a socket-based implementation. We are modifying the X library to buffer requests in shared memory. Flushes transfer the data to the shared buffer, and only cause an IPC message to be sent if previous IPC messages have not yet been collected.

5 All Services Do Not Need a Hardware Firewall

The cost of accessing functions in another protection domain (whether in the kernel or in a server's address space) has provided motivation for microkernel-based systems to migrate what were once kernel functions into client address spaces. This is possible when unprotecting the functions has no security implications. For example, there's no reason to put the system clock in the kernel, let alone another address space – on most architectures, the clock can be mapped directly into user address spaces. Similarly, network protocols can execute in the address spaces of the applications which are communicating rather than in the kernel or a special protocol server [Schroeder & Burrows 90, Maeda & Bershad 92]. Applications can send and receive packets directly through the network interface. If the network is assumed to be insecure (as is generally the case), then executing the protocols in a secure protection domain offers no additional integrity. Encryption, and not hardware protection modes, are necessary here.

Another technique for removing hardware firewalls is to migrate pieces of the operating system service into clients' address spaces. Mach's Unix emulation package uses this approach with its transparent emulation library, which is a shared library mapped into every Unix address space. Unix system calls are reflected out of the kernel into the caller's emulation library. There, the emulation library may simply forward the system call onto the Unix server, or it may implement the call itself, if possible. For example, the emulation library uses a mapped file interface for communicating with the Unix server. Read and write system calls are intercepted by the emulation library and converted into loads and stores to the mapped memory which backs the file. In this way, binary compatibility with Unix is maintained, and cross-address space IPC is avoided. This approach has been generalized in the Mach multiserver project [Julin et al. 91], and has resulted in substantial IPC reductions. In benchmarks on that system, which are intended to be persuasive but not conclusive, client-side emulation permits two out of three system calls to be implemented without an RPC.

6 Conclusions

IPC performance has come a long way in the last ten years. We now understand how to build IPC mechanisms which are only a few tens of microseconds slower than system calls. While this may at first seem unacceptably high, an examination of the other issues in operating system performance reveals that the additional overhead is small compared to the services which are being accessed. Moreover, the growing mismatch between cache and memory speed is making the physical location of operating system code and data much more important than the software path by which it is accessed. Finally, as microkernel-based operating systems have matured, useful techniques which reduce the frequency of operating system interaction, and hence, IPC, have also been developed. For these reasons, the raw performance of IPC facilities is largely becoming

an irrelevant metric by which to judge microkernel-based operating systems.

Acknowledgements

Jose Carlos Brustoloni, Rich Draves, Dan Julin, Mark Stevenson, and Bob Wheeler supplied most of the measurements presented in this paper. Conversations with these people, as well as with Bob Baron, Joe Barrera, Alessandro Forin, Michael Ginsberg, Chris Maeda and Dan Stodolsky contributed to this paper.

References

[Accetta et al. 86] Accetta, M. J., Baron, R. V., Bolosky, W., Golub, D. B., Rashid, R. F., Tevanian, Jr., A., and Young, M. W. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–113, July 1986.

[Agarwal et al. 88] Agarwal, A., Hennessy, J., and Horowitz, M. Cache Performacne of Operating System and Multiprogramming Workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[Anderson et al. 91] Anderson, T., Levy, H., Bershad, B., and Lazowska, E. The Interaction of Architecture and Operating System Design. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 108–121, April 1991.

[Bershad 90] Bershad, B. N. *High Performance Cross-Address Space Communication.* PhD dissertation, University of Washington, Department of Computer Science and Engineering, Seattle, WA 98195, June 1990.

[Bershad et al. 90] Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990. Also appeared in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989.

[Bershad et al. 91] Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M. User-Level Remote Procedure Call. *ACM Transactions on Computer Systems*, 9(2):175–198, May 1991.

[Bershad et al. 92] Bershad, B. N., Forin, A., and Draves, R. Cache Effects for a Microkernel Operating System. Technical report, School of Computer Science, Carnegie Mellon University, 1992. In preparation.

[Cheriton 84] Cheriton, D. R. The V Kernel: A Software Base for Distributed Systems. *IEEE Software*, 1(2):19–42, April 1984.

[Cheriton et al. 90] Cheriton, D. R., Whitehead, G. R., and Sznyter, E. W. Binary Emulation of Unix using the V Kernel. In *Summer 1990 Usenix Conference Proceedings*, 1990.

[Draves 90] Draves, R. P. A Revised IPC Interface. In *Proceedings of the First Mach USENIX Workshop*, pages 101–121, October 1990.

[Draves et al. 91] Draves, R. P., Bershad, B. N., Rashid, R. F., and Dean, R. W. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 122–136, October 1991.

[Golub et al. 90] Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an Application Program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–95, June 1990.

[Julin et al. 91] Julin, D. P., Chew, J. J., Stevenson, J. M., Guedes, P., Neves, P., and Roy, P. Generalized Emulation Services for Mach 3.0: Overview, Experiences and Current Status. In *Proceedings of the 1991 Usenix Mach Workshop*, November 1991.

[Leffler et al. 89] Leffler, S., McKusick, M., Karels, M., and Quarterman, J. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, Reading, MA, 1989.

[Maeda & Bershad 92] Maeda, C. and Bershad, B. N. Networking Performance for Microkernels. In *Proceedings of the Third Workshop on Workstation Operating Systems*, April 1992.

[Mogul & Borg 91] Mogul, J. and Borg, A. The Effect of Context Switches on Cache Performance. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, April 1991.

[Ousterhout 90] Ousterhout, J. K. Why Operating Systems Aren't Getting Faster As Fast As Hardware. In *Proceedings of the summer 1991 USENIX Conference*, pages 247–256, June 1990.

[Ousterhout et al. 88] Ousterhout, J., Cherenson, A., and Douglis, F. The Sprite Network Operating System. *IEEE Computer Magazine*, 21(2):23–36, February 1988.

[Rashid et al. 91] Rashid, R. F., Malan, G., Golub, D., and Baron, R. DOS as a Mach 3.0 Application. In *Proceedings of the 1991 Usenix Mach Workshop*, pages 27–40, November 1991.

[Rozier et al. 88] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Giend, M., Guillemont, M., Herrmann, F., Leonard, P., Langlois, S., and Neuhauser, W. The Chorus Distributed Operating System. *Computing Systems*, 1(4), 1988.

[Schroeder & Burrows 90] Schroeder, M. D. and Burrows, M. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[Wiecek 92] Wiecek, C. A Model and Prototype of VMS Using the Mach 3.0 Kernel. In *Proceedings of the 1992 Usenix Microkernel Workshop*, April 27–28 1992. This issue.

Fast Thread Management and Communication Without Continuations

Jochen Liedtke

German National Research Center for Computer Science (GMD)
5205 Sankt Augustin
Germany

jochen.liedtke@kmx.gmd.dbp.de

Abstract:
[Draves et al. 91] described some Mach kernel optimizations based on continuations resulting in e.g. 14% faster communication and 85% less kernel stack space. The present paper was stimulated by their work and presents results in some way contradicting it. Like the paper cited this paper is purely technical, not conceptual.

The main thesis is that highly efficient thread management and communication in an operating system must (and can) be obtained by a thorough and comprehensive design, not by one special trick. The arguments come from the operating system L3 [Liedtke et al. 91, Liedtke 92].
A null RPC in L3 executes 8.9 times faster than in Mach MK40 and exception handling 11.6 times faster. (L3 as well as Mach were measured on a Toshiba 5200 laptop with a 386 CPU, 20 MHz clock, 32K cache, 8M memory). These large factors can hardly be explained by slight differences between the Mach and L3 IPC semantics. A second reason for the factor coming mainly from implementation and not from different kernel concepts is the fact that the L3 kernel performance itself was significantly improved by a redesigned thread management and IPC implementation. It is 4 to 5 times faster than the old L3 version 2 implementation.
To date there exist about 300 commercial L3 installations.

1. Why efficient IPC?

Fast IPC will support all client/server architectures. Examples are standard operating systems emulated on top of a microkernel, file servers, replication servers ... and device drivers. The last aspect is not yet widely discussed, but is a good example for illustrating the performance requirements.

Nearly all arguments supporting dekernelization and the microkernel approach also demand implementing device drivers as user processes outside the kernel. So L3 was one of the first systems (perhaps the first practically available one), which strictly handled all devices (except the clock interrupt) outside the kernel by user level processes. The method used is simply transforming hardware interrupts into interrupt

messages and extending a task's address space by hardware I/O ports [Liedtke et al. 91]. Of course, flexibility and adaptability of the system increased. Furthermore we found it very nice to program device drivers like other servers using the kernel's abstraction and protection mechanisms. Usually such a driver program looks like

```
wait for first order ;
DO
     execute order ;
     reply and wait for next order
OD .
```

Now let us have a closer look at a typical PC's disk driver. The standard ESDI or AT−bus controller operates on 512 byte sized sectors and generates an interrupt after each sector read or written. Thus a disk read operation refines *execute order* mentioned above to

```
execute read disk order :

     setup read operation ;
     FOR i := 1 TO sectors to be read DO
          wait for ready interrupt ;              ◄ hardware RPC occurs
          read data
     UNTIL error occurred OD .
```

This can be interpreted as a remote procedure call (RPC) by hardware. The interrupt line sends an interrupt message to the device driver, which handles the interrupt and replies by continuing the interrupted thread. In fact such a hardware RPC costs about the same as the well known software null RPC.

Today's disks will deliver a sector every 260 μs (3600 rpm, 64 sectors per track) or even every 140 μs (5600 rpm, 76 sectors per track), i.e. one RPC every 260 or 140 μs. Obviously in this case you cannot tolerate 500 μs or more for a null RPC, except perhaps in a pure research system. Similar situations arise when using standard PC hardware for RS232: one interrupt every 520 μs (19.2 KBaud) down to every 86 μs (116 KBaud).

On the one hand it is true that increasing CPU speed may moderate the IPC performance requirements, but on the other hand the increasing speed of external devices (faster rotating disks, high speed networks) and multi media applications may require even higher RPC bandwidth.

In general, cheap RPC will lead to high acceptance of the concept by application program developers, whereas high costs will seduce programmers to circumvent RPCs by more or less dirty tricks, thus inducing bad, errorprone and unclear program structures. (Too often I have heard system programmers developing software on top of a microkernel talking about how to avoid context switches and IPC.)

Therefore one important goal of the L3 kernel redesign was to explore how fast IPC

can be realized. The results, e.g. 60 μs (386, 20 MHz) or 15 μs (486, 33 MHz) for a null RPC, are presented in section 2, whereas section 3 contains a critical discussion of Mach's continuation approach [Draves et al. 91] in relation to the L3 experiences.

2. The L3 Implementation

2.1 Threads, Tasks and Communication

The L3 micro kernel implements tasks consisting of an address space (into which memory objects can be mapped) and threads. Both tasks and threads are persistent objects. Actually the kernel supports up to 16383 threads per station. Threads communicate directly by messages which can be composed by strings and memory objects. Strings are transferred by strict copy whereas transfer of (mapped or unmapped) memory objects is done lazily (copy on write). The micro kernel manages tasks (including memory and address spaces) as well as threads (including communication). Memory objects are handled by the default or external pagers. I/O drivers are built as user level tasks or threads, located outside the kernel. The kernel transforms hardware interrupts into null messages to the associated threads.

IPC primitives are

```
send       (dest, msg, snd timeout, res)1)
receive    (source, msg, rec timeout, res)
wait       (msg, source, rec timeout, res)

call       (dest, snd msg, snd timeout, rec msg, rec timeout, res)
reply wait (dest, snd msg, snd timeout, rec msg, src, rec timeout, res)
```

Send does not block, whereas *call* and *reply wait* are blocking operations. *Receive* and *call* accept only messages from the source/dest specified, whereas *wait* and *reply wait* accept messages from any source. In fact more than 95% of the IPC in an L3 system is handled by either *call* or *reply wait*, because these two allow a simple and efficient RPC. Different to Mach no ports are involved. Direct access via thread ids, which are unique in time, saves bookkeeping and one indirection, but on the other hand requires validity checks per message transfer.

An IPC related concept unique to L3 is the *clan* [Liedtke 92]. This is a set of tasks headed by a *chief*. Inside the clan all messages are transferred freely and the kernel guarantees message integrity. But whenever a message tries to cross a clan's borderline, regardless whether outgoing or incoming, it is redirected to the clan's chief. This chief may inspect the message (sender and receiver as well as contents) and decide whether or not it should be passed to the destination to which it was

1) Timeouts are given in milliseconds; a special value denotes *infinite* or *never*. The operation's result *res* can be *ok*, *send/rec timeout*, *dest/source not existent*.

addressed. Obviously subject restrictions and local reference monitors can be implemented outside the kernel by means of clans. Since chiefs are tasks at user level, the clan concept allows sophisticated user definable security checks as well as active control, e.g. for debugging, emulating environments and forwarding messages via networks.

2.2 Performance Results

For measurement two hardware platforms were used: first a Toshiba 5200 PC with a 386 cpu (20 MHz clock) and 32 Kbyte cache, second a no-name PC with a 486 cpu (33 MHz clock) and 64 KByte secondary cache. (The 486 chip already contains an 8 KByte "primary" cache.)

To measure the RPC costs two user level threads running in different address spaces "playing ping-pong" were used. Table 1 contains the RPC times for various parameter sizes. Each time applies to a complete round trip[2)], where "ping" sends m bytes to "pong", which replies with n bytes. n and m are net message sizes, i.e. information about sender/receiver is added.

cross address space RPC $m + n$ parameters (bytes)	Toshiba 5200 386, 20 MHz	no-name PC 486, 33 MHz
0 + 0	60.20 μs	14.96 μs
8 + 8	60.20 μs	14.96 μs
12 + 8	74.94 μs	18.30 μs
128 + 8	81.14 μs	21.06 μs
512 + 8	101.74 μs	30.04 μs
12 + 12	89.64 μs	21.72 μs
128 + 128	102.02 μs	27.18 μs
512 + 512	142.92 μs	45.18 μs
exception handling within same address space	45.18 μs	11.44 μs

Table 1: RPC and exception handling times

This is far better than what most former microkernels have achieved, and it is fast enough to handle disks, RS232 and ethernet hardware; but is it close enough to the optimum?

[2)] It is assumed that the message will not cross a clan's borderline. Timeout is *never*.

386/486 processors have a special machine instruction which does a complete context switch including TLB flush. Two of these operations cost about 38 μs (386, 20 MHz) or 13 μs (486, 33 MHz)[3]. (You are right: this operation is not used in the L3 kernel.) Comparing this with the costs of a null RPC (60 / 15 μs) it seems reasonable that further significant gain of efficiency requires a conceptual change.

2.3 Implementation Methods

From the beginning we felt that good performance must be founded on a simple architecture and that highly efficient thread management and communication requires a thorough and comprehensive design on

- the kernel interface level,
- the algorithmic level,
- the coding level.

To illustrate the impact of various design decisions on each level, in the following the respective costs must be considered in relation to the resulting time for one short message transfer[4], 30 μs or 7.5 μs (for 386, MHz or 486, 33 MHz).

On the *kernel interface level* the system calls *call* and *reply wait* allow reduction of one system call per message transfer. (Switching from user into kernel mode and back to user mode costs 10 μs or 3 μs.) IPC based on *send;receive* communication would use two system calls and sometimes prevent direct transfer of flow control due to lacking atomicity.
To support efficient use of the IPC mechanisms by application programs, *dest, source, send message, receive message* (buffer) and *timeout* are independent parameters of the system calls. By this and the fact that messages can contain indirectly addressed strings unnecessary copy operations for composing or decomposing messages are avoided.

On the *algorithmic level* the most effective properties are:

- The L3 kernel strictly uses the process model, i.e. a kernel stack, which is part of the thread control block (tcb), is associated with each thread. (Actually a tcb with its kernel stack occupies a 4K page. Theoretically this can be reduced to 1K, even when using the 386 MMU.) The space of all thread control blocks is mapped into a special part of each task's address space, which is only kernel addressable. Thus accessing the actual as well as other tcbs is very simple and fast. Switching the context to another thread is mainly reduced to reloading the stack pointer (1 μs or

[3] These are the minimum times calculated from [i386] and [i486]. It is assumed that memory is accessed without wait states and only the necessary 5 TLB misses occur after a context switch (one page for code, stack, TSS, LDT and segment descriptors).

[4] This is one half of an RPC.

0.4 μs)[5] and switching the address space, if necessary. The facts that threads can block in kernel mode and tcbs can be paged (threads are persistent!) are simple consequences of this architecture.

- String messages are copied directly from the sender's to the receiver's address space by fast dynamic mapping of the destination area into the sender's address space.
- We found that 50% to 80% of the transferred messages were not longer than 8 bytes. Thus short messages are transferred via the CPU registers, similar to [Cheriton 84]. This is 1.5 times (15 μs or 3.5 μs) faster than copying from memory to memory.
- Whenever possible control is transferred directly. The sending thread simply switches to the receiving thread's kernel stack and address space (if necessary) and continues this one by returning to the user level. As a consequence a client automatically donates the current timeslice to the server and gets it back together with the reply.
- Bookkeeping of ready-, waiting- and other queues is done lazily. (Strict updating[6] would cost at least 10 μs or 4 μs per message transfer, compared to 30 μs or 7.5 μs needed for the complete IPC operation.)

Additionally some CPU specific methods were used on the *coding level.* Besides the well known local code optimizing techniques, the most important are

- o alignment of tables to minimize TLB and cache misses,
- o rearrangement of basic blocks to avoid frequent jumps,
- o avoiding use of segment registers.[7]

2.4 Are the Results Applicable to Other Architectures?

L3 has been implemented on 386/486 based PC's. Discussing the possible effects when changing the hardware architecture is highly speculative, but our experiences would suggest the following:

- Similar results will be achieved on other single processor systems, provided the processor is of von Neumann type, has a conventional MMU supporting a large address space and at least two modes (user/kernel) with own stacks.

5) Assuming that neither floating point nor debug registers are actually used by both threads.

6) Assuming that the queues are implemented as double linked lists.

7) Segment registers are a 386/486 specific feature.

- In a shared memory multiprocessor system locks must be used for synchronization. Even in the nonconflicting case this will slow down IPC. On a 386/486 based multiprocessor system with an efficient cache and memory architecture, we expect degradation of 10% to 20% for a null RPC assuming that client and server use the same processor and no conflicts occur.
- In contrast to the LRPC results [Bershad 90] message transfers between threads running on different processors, which can communicate only via memory, will cost substantially more than transfers between threads using the same processor. Only with transputer-like architectures inter processor communication can be made as efficient as intra processor communication.

3. Discussing Continuations

[Draves et al. 91] introduced continuations in the Mach kernel as a general optimization technique saving space and time. Since threads must be able to block in kernel mode, either one kernel stack per active thread must be allocated or an auxiliary data structure (per thread) called "continuation" is required. This is used to store all information necessary for continuing the thread later on. Thus the currently used kernel stack becomes available for the next thread. As a consequence, in most cases only one kernel stack per processor is needed and space is saved, because continuations can be dimensioned smaller than stacks.

Draves et al. mention that a null RPC executes 14% faster and exception handling performance has improved by a factor of 2 due to the use of continuations. But as shown in section 2, solutions without continuations are possible, which are about 10 times faster than Mach's continuation based implementation. Now suppose you combine the continuation technique with the L3 approach. What would be the effects, besides some loss of elegance and simplicity in the programming model?

3.1 Speed Detoriation

Positive or negative effects of continuations concerning execution time come out of

- optimizations due to continuations,
- runtime costs of continuations,
- change of working set (influencing TLB and cache misses).

The results presented in section 2 strongly suggest that using continuations in L3 will *not enable further optimizations.*

Estimating the *lower bound of runtime costs* can be based upon a best case analysis of the additional operations required for implementing continuations. At least copying the return vector from the stack into the continuation and recovering it later on is needed.

This overhead can be ignored as long as IPC is slow, but it may become relevant in connection with a fast IPC implementation. So we added saving/recovering of the return vector (5 words pushed by hardware) and one status word to L3's context switch operation and measured again the costs of a null RPC:

	null RPC time		increase
	L3	L3 + save/recover	
386, 20 MHz	60.20 μs	87.40 μs	+ 45 %
486, 33 MHz	14.96 μs	19.68 μs	+ 32 %

Table 2: Costs of save/recover continuation

At a first glance you would expect continuations reducing the *working set*, thus slightly increasing speed by better TLB and cache hit rates. But a closer look shows that no speed up can be expected:

- Without continuations kernel stack and thread control block can occupy the same page; with continuations two different pages are needed. Since the 386/486 has a "one process TLB", continuations lead to one additional TLB miss per context switch. (A "multi process TLB" would avoid this.)
- Since the most frequently used L3 kernel operations use only as much of the kernel stack as has also to be saved in the continuation, there is no reason for a significant cache working set difference between *n stacks* and *one stack plus n continuations.*

As a result we are convinced that introducing continuations in L3 will have no positive effects concerning speed, probably even decrease it.

3.2 Space Improvements

Since memory is not critical in a typical L3 application, we use one 4 K page for each thread control block including its kernel stack. But if necessary, a control block including stack could be reduced to 1 K. Although the 386/486 MMU operates with 4 K pages only, 1 K kernel pages can be realized by a special trick [Liedtke 92a], thus achieving a kernel working set of 1 Kbytes per active thread.

A continuation must be able to store at least the processor's general registers, floating point, segment and debug registers. Together with other kernel data in the thread control block about 500 bytes are needed.

So the use of continuations would save about 500 bytes per active thread.

This result is positive, but expressed in today's memory costs we are talking about 2 cents per active thread. (In most cases additional memory can easily be plugged into a computer.) Furthermore this gain has to be compared with the user level working set per thread.

4. Conclusions

- IPC can be implemented really fast.
- Continuations will not support this job.

References

[Bershad 90] Bershad, B.N.
High Performance Cross Address Space Communication
PhD dissertation, University of Washington, Seattle 1990

[Cheriton 84] Cheriton, D.R.
An Experiment using Registers for Message Based Interprocess Communication
Operating Systems Review 4/1984

[Draves et al. 91] Draves,P.D., Bershad,B.N., Rashid,R.F., Dean,R.W.
Using Continuations to Implement Thread Management and Communication in Operating Systems
in Proc. 13th ACM Symosium on Operating System Principles, Oakland 1991

[i386] Intel Corporation
80386 Programmer's Reference Manual
Santa Clara, 1986

[i486] Intel Corporation
i486 Processor Programmer's Reference Manual
Santa Clara, 1990

[Liedtke et al.91] Liedtke,J., Bartling,U., Beyer,U., Heinrichs,D., Ruland,R., Szalay,G.
Two Years of Experience with a Micro Kernel Based OS
Operating Systems Review 2/1991

[Liedtke 92] Liedtke,J.
Clans & Chiefs
Proc. 12. GI/ITG Fachtagung "Architektur von Rechnersystemen",
Berlin, Heidelberg, New York, Tokio, to appear

[Liedtke 92a] Liedtke, J.
A Method to Use Smaller Pages Than the MMU Allows
internal short note

Experience with SVR4 Over CHORUS

Nariman Batlivala, Barry Gleeson, Jim Hamrick,
Scott Lurndal, Darren Price, James Soddy
Unisys Corporation, San Jose, California

Vadim Abrossimov
Chorus Systèmes, Paris, France

Abstract

While many have touted the advantages of microkernel operating system architecture, there are few examples of complete, commercial quality systems built in this style. This paper describes our experience building a UNIX® System V Release 4 (SVR4) compatible operating system using the CHORUS® microkernel.

For this technology to gain acceptance in the marketplace, it must provide full binary compatibility with existing operating systems, and performance comparable to traditional, monolithic operating systems. In addition, it must allow developers to track the evolution of UNIX at a cost not much greater than a port, and it must enable the development of new system functionality.

1. Introduction

Microkernel technology is attractive for many reasons. Until recently, however, investigation of this technology was largely confined to academic settings. Wide acceptance in the commercial world requires solutions not only to the abstract problem of providing appropriate operating system services, but also to the practical problems faced by system vendors each day.

1.1 Requirements

Any commercial version of the UNIX operating system developed today must provide absolute conformance to the source compatibility standards and total binary compatibility with other platforms using the same processor. Since we initially chose the Motorola 88000 as our hardware platform, we have a stringent set of compatibility standards and compliance tests with which to conform. The 88open Binary Compatibility Standard (BCS) defines the trap interface for applications based upon SVR3.2 interfaces, and the 88open Application Binary Interface (ABI) defines the interface for those applications based on SVR4. Price/performance is another key issue. We cannot afford an operating system that prevents us from competing in this area. At the same time, we are prepared to absorb some performance overhead, particularly in non-critical areas, to ease the development of new features and functionality.

The ability to track the evolution of UNIX is at least as important as providing binary compatibility and competitive price/performance. It appears that UNIX is becoming more modular. It is commonplace today for I/O devices to be designed for industry standard busses, and for the device drivers to be provided by the manufacturer of the I/O device. Thus, source compatibility with the **Device Driver Interface** (DDI), enabling the direct importation of third party device drivers and STREAMS modules is a key requirement.

® UNIX is a registered trademark of UNIX System Laboratories, Inc. in the U.S.A. and other countries.

® CHORUS is a registered trademark of Chorus systèmes.

The **Virtual File System** (VFS) [Kleiman 86] interface has enabled the development of new file systems independent of the rest of the UNIX kernel. Hence, adherence to the VFS interface is also important.

We must be able to track future UNIX System Laboratories (USL) releases such as SVR4 ES and ES/MP, with a cost not significantly greater than that of a simple porting effort. This is a major challenge!

We also have an internal goal: this operating system should be an appropriate base for experimentation with and development of distributed systems. During design and development, we devoted considerable thought to the problem of presenting a "Single System Image" to users of a collection of machines running this operating system.

The target hardware for this operating system is the Unisys S/8400, a 33 MHz Motorola 88000-based machine with one or two processors, and a VME based I/O subsystem.

1.2 CHORUS Concepts

The abstractions provided by the CHORUS microkernel or **Nucleus** are similar to those found in other systems, such as Amoeba [Tanenbaum 90], Mach [Golub 90], and the V Kernel [Cheriton 90]. These abstractions include

- **Unique Identifiers**: global names used to provide a unified name space for all objects within a distributed CHORUS system. Unique identifiers (UIs) are guaranteed to be unique over time for all sites within an administratively designated domain.
- **Actors**: collections of resources within a CHORUS site, representing private memory contexts that may contain multiple memory regions, communication ports and threads of execution.
- **Threads**: sequential flows of control within an actor. All threads of an actor share all resources of the actor. Threads are scheduled independently by the Nucleus. Threads synchronize with each other using Nucleus provided **mutex** and **semaphore** operations.
- **Ports**: location transparent communication end-points within a CHORUS system. (A CHORUS system consists of a set of CHORUS sites.) Threads send and receive messages on ports, which serve as globally-named message queues. Ports are initially attached to a specified actor but can be migrated to another actor. Only threads within the actor to which the port is attached have the right to receive messages on it. Knowledge of a port's name implies the right to send messages to that port.
- **Port Groups**: collections of ports that are addressed as a group to perform communication operations. CHORUS port groups provide several addressing modes. Messages may be sent to all members of the group, any one member of the group, one member of the group residing on a particular site, or one member of the group residing on the same site as another known port.
- **Messages**: untyped sequences of bytes that represent information to be sent from one port to another via CHORUS IPC. All messages transmitted are stamped with the protection identifiers of the sending actor and port.
- **Regions**: contiguous ranges of valid virtual addresses within an actor. Each region is treated as a unit by the CHORUS virtual memory system. Multiple regions within a single actor may not overlap.
- **Segments**: encapsulated data within a CHORUS system, typically representing some form of backing store, such as a swap area on a disk. Regions map a portion of a segment into the address space of an actor. Requests to read or modify data within a region are converted by the virtual memory system into read or modify requests within the segment through the use of a standard Nucleus-to-mapper protocol. Any server adhering to this protocol can act as a **mapper** for virtual memory segments. Mappers also provide the synchronization needed to implement distributed shared memory.
- **Capabilities**: unique handles, the possession of which grants the right to perform an operation. Within a CHORUS system, capabilities consist of the concatenation of a port name and a key. The port name

identifies a server and the key identifies an object within the server.

Of the above abstractions, unique identifiers, actors, threads, ports, port groups, messages, and regions are defined and managed solely by the CHORUS Nucleus. Capabilities and segments are managed cooperatively by the Nucleus and system servers. For further information concerning the CHORUS Nucleus abstractions, see [Rozier 88].

A distinguishing characteristic of the CHORUS Nucleus is that the actor abstraction was extended by the introduction of the **supervisor actor.** Supervisor actors are much the same as other actors in that they are compiled, linked, loaded into memory, and unloaded from memory separately from the Nucleus and from each other. They differ from other actors because, when loaded, they execute in a privileged instruction mode and share the system address space.

Supervisor actors enable several optimizations that greatly enhance the performance achievable in a modular subsystem. The two most notable optimizations are the ability to use **connected handlers** and an extremely fast lightweight RPC mechanism.

Connected handlers allow efficient treatment of hardware events. Using an architecture-independent interface, supervisor actors may associate functions dynamically with traps, exceptions and interrupts. The use of connected handlers eliminates the requirement that device drivers be part of the microkernel, without sacrificing performance.

The CHORUS lightweight RPC mechanism (LWRPC) optimizes communication between supervisor actors that reside on a single site. When a LWRPC can be performed, the cost of communicating between the two supervisor actors is slightly more than that of a subroutine call. This optimization is performed automatically by the Nucleus, transparent to the client of the service.

Using these abstractions, a high-level operating system interface, such as that provided by UNIX, is exported. Such an operating system interface is called a **subsystem**. Typically, subsystems consist of several multithreaded system servers, each providing a part of the exported interface. The protocols between the system servers are designed so that the operating system interface can be readily extended to a distributed environment. Each remote request to a server is processed by one of its threads. Part of the request message contains information needed to construct the context within which to process the request.

Each server creates one or more ports to which remote clients send requests. Some of these ports may be inserted into port groups with well-known names. Such port groups can be used to access a service independent of the provider of the service.

Servers may also provide a trap interface to services. These interfaces can be compatible with existing UNIX-like system interfaces, but do not extend transparently across the network. To provide a distributed system interface while maintaining binary compatibility, system calls are normally handled by a server that exports a trap interface. System calls that reference remote entities are marshaled into messages and sent using CHORUS IPC to the server that manages the remote entity.

1.3 Architecture of SVR4 over CHORUS

The operating system consists of up to five distinct servers, depending upon the needs of a particular site. (See Figure 1.) These subsystem servers are

- Process Manager (PM) - manages process contexts and dispatches system calls.
- Object Manager (OM) - manages file systems, block device drivers and non-STREAMS character drivers.

- STREAMS Manager (STM) - manages STREAMS drivers, pipes and all other STREAMS dependent services.
- IPC Manager (IPCM) and IPC Key Manager (KM) - manage System V IPC (shared memory, semaphores and message queues) and their associated keys, respectively.

The servers are known collectively as the **SSU**.

This division into servers is similar to that used in CHORUS/MiXV3.2, a version of UNIX System V 3.2. The IPCM and KM extend the operating system services to include all System V IPC facilities. The Device Manager of MiX V3.2 was replaced in the SVR4 implementation by the STM. In addition, file data is cached by the Virtual Memory subsystem, which is managed by the Nucleus. This required the development of significant new protocols between the OM, PM and Nucleus.

The OM, STM, IPCM and KM use large sections of unmodified SVR4 code, and are written in C. The Nucleus and PM are written in C++.

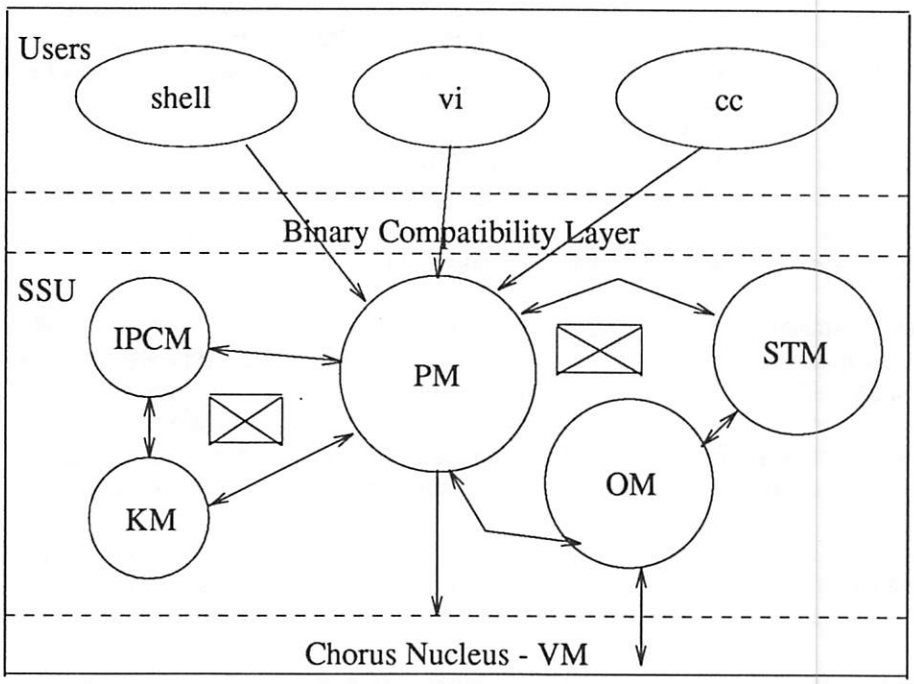


Figure 1. SVR4 Servers

2. File System

SVR4 introduced new file system services, and a major file system reimplementation. Among the new services are *mmap*(2) and enforcement mode record locking, which pose some interesting problems in a distributed environment, such as that provided by the CHORUS Nucleus. Since *mmap*(2) permits simultaneous access to file data through the load and store operations, as well as the through traditional *read*(2) and *write*(2) system calls, the USL implementation features elaborate interactions between the file system code and the Virtual Memory code.

The CHORUS model calls for a separation between the client Nucleus (or user process) and the mapper. Implementing this separation means that the Nucleus and the OM do not share data structures. We describe here the mechanism we have chosen to provide the services and semantics of SVR4 files while maintaining the model of client and server separation.

2.1 Memory Mapped Files and the Page Cache

A typical UNIX system call for file system services begins as a trap into the PM. In the case of *open*(2), a message is sent to an appropriate OM determined by the pathname of the file. If the open is for a cacheable (mappable) file, the open will return a special **capability** which is used to locate the mapper (OM) and identify the file to the mapper. The following sections describe the how reading and writing data is done through the Nucleus page cache.

2.1.1 Read. When a *read*(2) system call is initiated on a cacheable file, a *sgRead*(K) call is made to the Nucleus. The arguments include the capability mentioned above. Using this capability, the Nucleus can send an appropriate message to the OM for that file.

In the simplest case, the Nucleus is asked for data that is already available in the page cache. In this case the data is provided with no file system interaction. If the Nucleus is asked for data that is not already cached, a **PULLIN** message is sent to the OM.

If the file is mapped, the PM creates a region in the process backed by the file. A Nucleus call sets up the address space information and provides the capability obtained when the file was opened. A fault for read in this process is handled much as the read cases described previously. Specifically, if the page is not already cached, a PULLIN message identical to the one previously mentioned is sent to the mapper.

Note that the role of the OM for *mmap*() is trivial. The OM must know that mappings exist, because they represent a reference to the vnode and because they must be synchronized with mandatory locking. So minimally, the OM must be told when the first mapping is created, and when the last one is destroyed.

2.1.2 Write. Write is handled similarly to read, but we must consider also the mechanism for maintaining the atomicity and serializability expected for UNIX file system operations. When a *write*(2) system call is initiated on a cacheable file, a *sgWrite*(K) call is made to the Nucleus. As before, the capability required to send a mapper message is included.

Consider the case that the write is for exactly one full page of bytes for a new file. The Nucleus has no requirement to obtain data from the mapper, but the file system must allocate disk space for the data to be written. In this case, a **GETACCESS** message is sent to the mapper asking for **write access** for the correct offset and size. This gives the mapper the opportunity to allocate disk space or to fail the request.

If the request is successful, and the file is not truncated before the next *sync*(2), the Nucleus eventually sends a **PUSHOUT** message to the OM with the new data.

Next consider the case that some bytes are modified in the middle of an existing file by the write. If the Nucleus has the corresponding page with write access, no interaction with the file system is necessary. If the Nucleus has the page with **read-only access**, the Nucleus must (as before) send a GETACCESS to the mapper asking for write access. This gives the mapper a slightly different opportunity than in the previous case. The mapper can determine that no other clients have read access to the page before granting write access to the requesting Nucleus. If the Nucleus does not have the page, it sends a PULLIN message to ask for the data with write access.

Note: In the read case above, the PULLIN message carries with it a request for read access to the relevant portion of the file. In the event of read/write activity on a file, the mapper similarly determines that there is no other client with write access, before granting the read access.

If a write fault occurs for a page of the same file, the interactions between the Nucleus and the mapper are the same as when the page is modified by a *write*(2) call.

2.1.3 Larger Reads and Writes. Consider the case of a read that spans 100 pages of file data. The Nucleus might be better off to execute a number of small PULLIN requests rather than one large request. I/O atomicity is maintained because the Nucleus may request, for example, the first two pages of data, along with 100 pages of read access with the first PULLIN message of the transaction. The Nucleus holds the read access until all 100 pages have been transferred. This eliminates the possibility of another client obtaining write access during the course of satisfying the *sgRead*(K). A write might similarly involve a large GETACCESS request.

It is an important aspect of this architecture that once access rights are held by a Nucleus for a portion of the file, the I/O may proceed without holding any locks on the file system. This permits us to perform simultaneous I/O for multiple processes on non-overlapping ranges within a file.

2.1.4 Flushing Data and Access Rights. The mapper may need to change the access that a client Nucleus has to all or part of a file. For example, if a file is truncated we must destroy all cached pages. This is accomplished with the *sgFlush*(K) call, which has options to destroy parts of a cache, to downgrade access from write to read, or simply to cause dirty pages to be flushed to backing store.

2.2 Attributes and Tokens

Since file I/O can take place ''behind the back'' of the file system, a mechanism is needed to keep attributes, such as file size, access time and modification time, current. The system call code in the PM is required to maintain these attributes, if they change as a result of a *sgRead*(K) or *sgWrite*(K) nucleus call. It is further necessary for the PM to ''know'' the size for such files, so that it can validate the size and offset of a *read*(2), for example. Thus, the PM maintains a client-side attribute cache.

We have implemented a shared reader, single writer token protocol for attribute mangement. The current state of the token is associated with the **vnode** for each cacheable file. When an OM receives the first open request for a file from a PM, the reply message will, if possible, indicate that the PM is receiving the readable token and file size as part of the reply. At the completion of the open, the OM and PM may both have an accurate size for the file.

If the PM is going to change the size of a file, it must first request the writable token from the OM. After granting the request, the OM may no longer know the exact size of the file. The OM, of course, always knows the amount of disk space allocated for use by the file. At sync time, the attributes that should go to disk are sent to the OM in the sync message.

If the OM needs to use or modify the size of the file, for example because of truncation, the OM must recall the appropriate token from the PM that holds it. While the readable token is not present, the PM may not use the size of the file.

The distribution of size information results in an inconvenience for file systems that have blocks size smaller than the system page size. When a small file is created and written, only the PM knows the exact size of the file. The Nucleus asks for write access to the first page of the file, and backing store must be allocated for the entire page. If blocks are allocated but not written, they must be freed when the vnode becomes inactive. In fact, the excess blocks are cleaned up on the last close of a file.

2.3 Mandatory Record Locking

The I/O path is direct from the PM to the Nucleus. The PM must, therefore enforce record locks. When the OM receives a request to place an enforcement mode lock on part of a file, each PM that has the file open must agree that the lock is in place before the OM can grant the request.

2.4 The Proxy File System

The implementation of STREAMS-based device files is separated into multiple servers. The name space for these files is managed by the OM, while the devices themselves (with their associated drivers) are

managed by the STM. This distribution is managed by a pseudo-file system, called the **proxy** file system.

While the PM-Nucleus-OM implementation of files with cacheable data is based on a model of client-side attribute caching in the PM, the proxy file system is based on a model of server-side attribute caching in the STM. Server-side attribute caching minimizes both complexity and messaging overhead for files whose name space management and data management are distributed across actors.

Proxy file system synchronization is controlled by the OM, but the file objects themselves (and their attributes) are managed by the STM. The proxy file system emulates the SVR4 **specfs** file system on the coordinator (OM) side, and emulates the underlying disk-based file system on the server (STM) side. The proxy server-side attribute caching model is implemented by using a coordinator-server protocol between the OM and STM.

When, during the course of a name space lookup operation, the OM recognizes the first open of a file object managed by the STM, the file object attributes are migrated to the STM with a **MOVEINODE** message, which creates a capability for the STM server file object. The OM proxy file system constructs a data structure (similar to a specfs vnode) to which the lookup operation resolves.

An open operation performed on the proxy file object sends a **LEAFOPEN** message to the server STM designated by the server capability. (The conceptual separation of the MOVEINODE operation and the LEAFOPEN operation is necessary to maintain control of the synchronization of opens and closes at the coordinator.) To complete the open operation, the coordinator (OM) returns a capability for the server (STM) file object to the client (PM). Subsequent operations on the open file exchange messages directly between the client (PM) and server (STM) with no intervention by the coordinator (OM).

Operations performed using the server file capability can be handled with a single message from client (PM) to server (STM). Using the server-side attribute caching model, implicit attribute modifications (those occurring as side-effects, such as modification time updates) can be handled at the server. Operations that access the file through the file system name space are still managed at the coordinator (OM). Since the coordinator's copy of the file attributes is not guaranteed to be up to date, an operation such as *stat*(2) requires a **GETATTR** message from the coordinator (where the name resolves) to the server to retrieve the most recent copy of the file attributes. Similarly, an operation such as *chmod*(2) (which also locates the file through the file system name space) requires a **SETATTR** message from the coordinator to the server to modify the attributes cached by the server. When a *sync*(2) occurs, the server (STM) locates all locally-managed file objects with modified attributes, and syncs the attribute information back to the coordinator (OM).

When the last reference to a server file object is removed, the server (STM) sends a **LEAFCLOSE** message to the coordinator (OM). Depending on the sequencing of opens and closes, the coordinator responds with either a **LASTCLOSE** (no other references outstanding) or with another LEAFOPEN (indicating that an open was in progress at the coordinator while the server was handling the close). The server responds to a LASTCLOSE message by releasing any local data structures for the server file object, and returning any modified attribute information to the coordinator.

The server-side caching model minimizes message traffic for operations that access or modify file attributes. Operations like *read*(2) and *write*(2) have the side-effect of updating timestamps for the file. Operations such as *fchmod*(2) explicitly modify attributes that, if the client-side caching model were used, would not be cached at the client. Operations such as *fstat*(2) access all attributes of a file. If the client-side caching model were used, some operations at the server would be required to fetch attributes from both the client and the coordinator. The server-side attribute caching model provides all the required functionality and distributed synchronization at minimum message overhead in the general case (with only slightly increased cost in the exceptional cases).

3. STREAMS and Device Management

3.1 STREAMS

We attempted to change the SVR4 STREAMS code as little as possible when integrating it into the CHORUS environment. To accomplish this, it was necessary to provide an environment for this code that included constructs such as a **user area**, a **proc structure**, and a **STREAMS scheduler** since such constructs do not exist in the Nucleus itself. In our implementation of UNIX over CHORUS, therefore, messages sent from the other actors to the STM must first execute a **wrapper** layer of code that creates an emulation of the necessary environment before the SVR4 STREAMS code is executed. The information necessary to create the environment (PID of the process making the request, job control disposition of that process, size of a read or write, etc.), is generally contained in the request message itself.

Some elements of the monolithic UNIX environment are too costly to send to the STM in every request message. (Process **credentials** are one such element.) To avoid this overhead, such pieces of the UNIX environment are generally cached in the STM and kept in a consistent state via protocols run between the STM and other actors. For example, although process credentials are actually created and modified in the PM, some are also cached in a read-only state in the STM. In each request message sent to the STM that requires process credentials, the PM instead passes a **credentials capability** that references those credentials. Upon receiving the request message, the STM uses this capability to search its cache of process credentials. If the credentials are not in the cache, the STM sends a request to the PM whose UI is stored in the credentials capability, asking for a copy of the credentials corresponding to the capability. (The cache size is limited by discarding cache entries when a predetermined threshold of unreferenced credentials is reached.) Thus, sending large data structures in every request message is avoided.

3.2 Scheduling and Interrupt Handling

Threads that enter the kernel as a result of executing system calls (as opposed to threads that do so as a result of an interrupt) are commonly referred to as **base level threads**, and are said to be running code at base level. As in previous versions of UNIX from USL, SVR4 does not allow preemption of processes running code at base level in the kernel, although there are **preemption points** in the kernel where processes can explicitly allow themselves to be preempted. Thus, SVR4 is essentially single-threaded at base level in the kernel. In contrast, threads executing in CHORUS actors may be preempted at arbitrary points in their execution. To provide the SVR4 code with the illusion of non-preemptability, the OM and the STM each manage their own **base level mutex**. Before running any kernel code in these actors from base level, the base level mutex must be acquired. The mutex is released when a thread leaves an actor, or when the thread invokes a DDI function such as *sleep*(D3DK) which explicitly allows for rescheduling. Thus, although both the OM and the STM preserve SVR4's single-threading at base level, it is possible for a thread in the OM and a different thread in the STM to be executing concurrently.

Since our operating system is DDI compliant, the SVR4 ***spl*** calls are supported. The mechanism used does not directly manipulate a hardware mask, however. *Spl* calls (such as *splhi*(D3D), *splx*(D3D)) allow base and interrupt threads in SVR4 to synchronize access to critical regions of code. In our operating system, calls to raise the processor *spl* level result in the acquisition of an actor's **spl mutex**. Lowering the *spl* level to 0 results in the release of the actor's *spl* mutex. Interrupts that are directed by the Nucleus to an actor first execute code which attempts to acquire the *spl* mutex. If successful, the driver interrupt handler executes immediately. If the *spl* mutex is not available, a request is queued for a **deferred interrupt thread** to acquire the mutex and run the interrupt handler later when the mutex is released. Thus, mutual exclusion between base and interrupt level threads accessing *spl* protected code regions is maintained.

3.3 Device Management

In SVR4, the Device Driver Interface (**DDI**) was introduced in an attempt to better define and isolate the interface between the kernel and device drivers. SVR4 provided some backward compatibility for drivers not written to conform to the DDI, but since we were developing an operating system for a new computer architecture and wanted to limit device driver access to kernel data structures, we decided to mandate DDI

conformance for all drivers used with our operating system. DDI conformance helps limit the amount of SVR4 environment emulation required of each of the actors.

For the majority of the DDI functions, we used the SVR4 code without change. The *datamsg*(D3DK), *putnext*(D3DK), and *qreply*(D3DK) routines are examples of functions that fell into this category. DDI functions related to virtual memory management (such as *vtop*(D3D)) were modified to interface to the Nucleus primitives, but were otherwise not difficult to port. For DDI functions related to synchronization between base and interrupt level threads (such as *splstr*(D3D), *splx*(D3D), *sleep*(D3DK) and *wakeup*(D3DK)), we chose not to use the SVR4 code, but rather to implement equivalent functionality with new code. In the case of *splstr*(D3D) and *splx*(D3D), this decision was driven by our desire not to have a global hardware mask, so that the OM and STM could handle interrupts independently and not interfere with one another. There were no Nucleus calls to do *sleep*(D3DK) and *wakeup*(D3DK) directly, so we implemented these functions using the Nucleus semaphore primitive[1]. The *physiock*(D3D) function, which sets up DMA between a device and a user's address space, was somewhat difficult to implement primarily because of the finite message size provided by the Nucleus. There were finally a few DDI functions (notably *copyin*(D3DK) and *copyout*(D3DK)) whose interface specification was actively hostile toward allowing them to function in a distributed environment. Some of the problems encountered porting these functions are discussed later in this paper.

4. Process Management

4.1 System Call Handling

The user's interface to the operating system is provided by two mechanisms. Applications compliant to the 88open BCS use trap instructions compiled into the executable file to invoke services from the operating system. Applications compliant to the 88open ABI link at run-time with a library that provides services. By using a trap interface from the library itself rather than the application, the ABI provides a greater degree of application binary portability while ensuring maximum flexibility for the operating systems system call implementation. Among the responsibilities of the PM is the handling of system call trap instructions.

The PM, as part of its initialization, requests the nucleus to establish **trap handlers** for the traps managed by the PM. When a thread executes a trap instruction to a managed trap vector, the nucleus executes the PM trap handler providing the trap number and the context describing the thread at the time of the trap. A value in a register at the time of the trap indexes into a vector containing function pointers for each system call.

Subsequent handling of a system call depends upon which subsystem actor provides the required service. Some system calls, such as *getpid*(2), can be serviced directly by the thread executing the trap handler. Most calls, however, require service from another actor or another thread within the PM. These calls can be roughly assigned to two functional areas: file system interfaces, such as *open*(2), *read*(2), *ioctl*(2), *mmap*(2), and process control interfaces, such as signal management, process creation and deletion, and retrieving process information. The file system interfaces result in a message being dispatched to the OM or STM for subsequent handling, while process control messages are sent to the control port (see Section 4.2 "Session and Process Group Management") of the target process.

4.2 Session and Process Group Management

SVR4 introduced the notion of process affiliations. A set of processes represent a user's login **session**; those processes may be subdivided into distinct **process groups**. Mechanisms have been provided to

1. Francois Armand of CHORUS provided our prototype implementation of *sleep*(D3DK) and *wakeup*(D3DK).

manage these affiliations. The user can create process groups that may be assigned as jobs, suspend and resume process groups, and move process groups into the foreground or background of the session's controlling terminal.

We have emulated SVR4 process affiliations and their management using unique identifiers (UIs), ports and port groups. Each process has a **control port** on which it receives various process control messages. For example, signals are sent to processes by sending a message to their control port.

SVR4 process identifiers (PIDs) are mapped to the UI of the process's control port by using *uiBuild*(K). Arguments to *uiBuild*(K) include the PID of the process and the UI type: K_UIPORT. Thus, servers such as the PM, OM and STM can fabricate the UI of a process's control port given its PID. Knowing the port UI then gives a thread the right to send messages to that port.

Similarly, we have mapped the notions of session and process group onto the Nucleus notion of port group. Processes become members of a session or a process group by inserting their control port into the port group for that session or the process group. Port group names, represented as capabilities, are also built by using the *uiBuild*(K) function.

The following examples illustrate the application of the ideas explained above:

4.2.1 Creating Sessions. Upon the successful completion of a *setsid*(2) system call, the calling process (a) is the session leader of a new session, (b) is the process group leader of a new process group, and (c) has no controlling terminal. The session ID and the process group ID are set to the PID of the calling process. Further, the calling process is the only process in the new session and the only process in the new process group.

To enforce these requirements, the following actions are taken:

- A port group UI for the process group is constructed from the calling process's PID, by using *uiBuild*(K). A message is broadcast to all members of the port group, excluding the calling process. If a process replies, thereby indicating that the port group (process group) already has a member, the session creation is not allowed and *setsid*(2) returns an error to the caller.

  If the *ipcCall*(K) returns a K_EUNKNOWN error, thereby indicating that the port group has no reachable member, session creation proceeds.

- Since the process is changing its process group and session affiliation, the process's control port must be removed from the current port groups and inserted into new port groups.

- The process's parent is notified of the change in affiliation by sending the parent a message on its control port.

4.2.2 Creating Process Groups. The system call *setpgid*(2) may be used to create a new process group within the session of the calling process. This requires the removal of the process from an existing process group and installation in a new process group. This is achieved by moving the process's control port from one port group to another.

Since process affiliations are changing within the session of the calling process, additional work is required to fix up the process group parent linkages. A process group parent inserts its control port into a port group named after the PGID of its child's process group. The process group parent port group also serves as an efficient mechanism for handling orphan process groups. This has greatly simplified the algorithms used in the distributed environment.

4.2.3 Joining Process Groups. The system call *setpgid*(2) may also be used to join an existing process group. This is a cooperative effort between the calling process and some member of the process group. The calling process requests membership in the process group by broadcasting to the target process group,

using functional mode to address any single member of the group. Once the request is granted, the calling process inserts its control port into the appropriate port group. Finally, the calling process informs its parent of the process group change by sending a message to the parent's control port.

4.3 Architecture of the `/proc` File System

Two actors are involved in `/proc` file system operations. The OM manages the file system mount point, virtual file system semantics, and open requests for files in the `/proc` file system. The PM acts as a server for the file system (much as the STM acts as a server for STREAMS-based files) handling requests such as *read*(2), *write*(2), *stat*(2), *ioctl*(2) and *close*(2).

Most operations against files in the `/proc` file system are handled in whole or in part by the PM. For example, the *open*(2) system call is sent initially to the OM for pathname analysis; when the `/proc` mount point is encountered by the OM during pathname resolution, a message is sent to the PM to resolve the rest of the path. The PM creates a local data structure associated with the file and returns a capability describing it to the OM. This capability includes the UI for the PM request port so that any further system calls against the file are routed directly to the PM.

4.3.1 Directory Operations The `/proc` file system is mounted at a file system mount point and as such may be the target of standard directory operations such as *opendir*(3), *readdir*(3), and so forth. This directory is constructed dynamically when a directory file is opened and read or when the vnode operation *VOP_READDIR*() is performed on behalf of an *open*(2) call.

4.3.2 Role of the Object Manager Pathname resolution is a function of the OM. Since `/proc` is implemented as a virtual file system, it uses standard pathname resolution facilities. This requires a vnode operations vector for `/proc` with an entry point known as *VOP_LOOKUP*(). When this entry point is called during pathname resolution, the OM will send a message to the PM. The PM searches its internal process list to resolve the leaf name.

A more common directory operation in the `/proc` file system is directory access via *readdir*(3). This involves opening the directory represented by the path `/proc`. Reads and writes on this file must be satisfied by the PM.

When a request to read the `/proc` directory is received, the OM forwards the request to the PM to obtain a buffer of device independent directory entries (in the format prescribed by the *X/Open Portability Guide* definition for `<dirent.h>`). The OM then copies the entries back to the requestor as appropriate to the size of the requestor's buffer and the current offset within the directory (as maintained by the standard offset management facility).

4.3.3 Role of the Process Manager The PM, as previously stated, acts as a server for `/proc` file system requests. The PM service code handles four types of requests: open/close operations, read/write operations, directory operations and ioctl operations.

Service requests to the PM are handled asynchronously with respect to the execution of the thread being managed, yet most requests are allowed only when the target process is stopped. To facilitate this, when a debugger requests a process stop (via an *ioctl*(2) of **PIOCSTOP**), the PM service thread handling the request first suspends the target thread. It then determines whether the thread is currently executing user mode code, or executing within the operating system processing a system call. In the former case, the thread is left suspended until the debugger explicitly causes it to resume execution (via **PIOCRUN**); in the latter case, a thread-local flag is set to cause the thread to stop itself just before it returns from the system call (in the PM trap handler).

Read and write operations are handled by using the Nucleus primitive *vmCopy*(K) to copy data to/from the stopped thread's address space.

Various control operations may be targeted to the thread by using the *ioctl*(2) system call. Some of the operations include retrieving process credentials, region mappings, *ps*(1) command data, registers, and signal actions. Control operations are also available to cause the target thread to stop on particular signal(s), fault(s) and/or system calls. When one of these events occurs, the thread stops itself and notifies the debugger by using a semaphore that it is stopped.

5. Experience

5.1 Serverization as a Development Aid

Microkernel-based software systems are generally built as collections of servers. The messaging interface between servers (or between microkernel and server) cleanly encapsulates the environmental data used by each operation in a transaction-like manner.

There is generally a crosstalk-induced limit on the number of developers that can effectively be applied to a single software subsystem. By reducing the grain of the subsystem from the entire operating system to an individual server, the overall size of the development team can essentially be increased proportionately to the number of server-level system components, with a corresponding decrease in elapsed development time.

If the messaging interface is sufficiently well-defined (as in, for example, the interface between the Nucleus Virtual Memory and the external mapper), development on the two ends of the interface may proceed independently. In our experience, such parallel development may involve large geographical separation with no significant loss of development time or effort. While it can be claimed that a similar level of interface definition and independent development can be achieved even in monolithic systems, there has historically been a substantial gap between what is theoretically possible and what is commonly practiced.

Our experience with the serverization of SVR4 has highlighted numerous modularity violations in the monolithic implementation. Some features of the operating system (such as controlling terminals, NFS, and /proc) required substantial rework to achieve a clean serverized implementation. The microkernel model lends itself naturally to the common practice of well-defined interfaces between system components and the parallel development of such components.

Functionality testing can often be reduced to verifying the messaging interface: the message request contains all of the relevant parameters and the message reply contains all of the relevant results. The server cannot reference global data structures outside the local address space, and the operation of the server can therefore be unaffected by such external data, nor can the external data be affected by the server.

Given that all of the state relevant to the execution of a particular operation is contained within the server and the message, it has been straightforward to implement server-specific debugging functionality as extensions of the general operating system debugging facilities. Like the implementation of the servers themselves, the debugging extensions are not visible to, nor do they have any effect upon, any data or state external to the server.

The microkernel interfaces are equally available to servers executing either in system mode or user mode. It is therefore feasible to verify a large part of the functional correctness of a server using user program development tools, given some small amount of emulation of the system environment. Assuming that the compile-and-execute cycle for the system-level development environment involves system reboots, user-level server development presents a considerable time savings. In particular, the provision for user-level external mappers was found to be a significant benefit.

5.2 Copyin/Copyout Emulation

Copyin(D3DK) and *copyout*(D3DK) are DDI functions responsible for moving data from/to the kernel address space to/from the user address space. Each of these functions takes three parameters: a user address, a kernel address and the length in bytes of the data to transfer. Implicit in the semantics associated with the invocation of these two functions is the assumption that there are two address spaces that can be implicitly located without passing any information through the interface to the function itself, namely, the address space of the current running user, and kernel address space. In monolithic SVR4, this assumption is, of course, met.

Our operating system, however, must be concerned with more than just the kernel and user address spaces. Each actor conceptually has its own address space. (In truth, supervisor actors located on the same site share the kernel address space for that site, but this optimization is not visible to them.) Additionally, our distribution goals required us to consider the possibility that a user process on one site might access a device managed by an actor located on another site. In this environment, a more explicit means of identifying the user address space in question was needed. **Actor capabilities** are provided by the Nucleus to name address spaces unambiguously. Thus, it was only necessary to make this information available to *copyin*(D3DK) and *copyout*(D3DK). Unfortunately, our strict DDI conformance constraints made the functional interface immutable, and so we chose to pass the actor capability to these functions via the emulated user area. In each request message sent to either the OM or STM that might result in either of these functions being called, the actor capability for the user process associated with the request is also sent. The **wrapper code** in the OM/STM puts the actor capability in the user area, where it can later be retrieved.

5.3 DMA Operations

Allowing DMA to be performed by devices to/from user memory is fairly straightforward in monolithic SVR4. (The DDI function *physiock*(D3D) is used by drivers to perform these DMA operations.) Both device and target user address space are located on the same machine. Since we wanted to design an operating system that could be distributed, however, we had to devise a means of supporting DMA in a way that did not require the device and the target user space to be co-located. We also had to take into account the finite message size provided by the Nucleus when implementing our solution. (The limit is approximately 64K in the version we run.) Finally, we did not want the solution that we implemented for distributed DMA to prevent the local DMA from working efficiently.

The two conditions that determine the mechanism used to accomplish a DMA operation in our operating system are

- Whether or not the DMA device and the target user address space are co-located.
- Whether or not the amount of data to be moved exceeds the maximum CHORUS message size.

In all request messages sent to the OM/STM that could result in DMA taking place (normally only messages sent as a result of a *read*(2) or *write*(2) system call), sufficient information is provided to allow the above two conditions to be determined. The co-location condition is determined by placing the **actor capability** for the target user actor in the request message, and having the *physiock*(D3D) function check this UI to see if it has been locally created or not.[2] Whether or not the amount of data to be moved exceeds the maximum message size is determined by putting the user buffer size for the *read*(2) or *write*(2) call into the request message for *physiock*(D3D) to examine.

2. Actor capabilities have information about the site where the actor originated embedded in their UIs. CHORUS provides a function (*uiIsLocal*(K)) to determine if a UI is local to a particular site.

If the DMA device and the target user address space are co-located, we perform the DMA operation in essentially the same way that SVR4 does, by locking down the user address range. If they are not co-located, then the action taken depends upon the DMA operation size. If it is smaller than the maximum message size, the request or reply message body is used as the DMA target buffer, depending on whether data is being transferred to or from the device, respectively. If the DMA operation size is greater than the maximum message size, a temporary buffer is allocated on the site that contains the DMA device, and the data is transferred between the user address space and the buffer using the *vmCopy*(K) Nucleus call, which allows arbitrary amounts of data to be moved between any two address spaces.[3] The transfer between the buffer and the DMA device takes place in the normal manner. Thus, we can support drivers that perform DMA, even though the device and the target address space are not necessarily co-located.

5.4 Controlling Terminals

The implementation of controlling terminals in monolithic SVR4 uses code in several normally disparate subsystems:

- The process management subsystem identifies session leaders that are allowed to allocate controlling terminals.
- The STREAMS subsystem determines which devices are terminals, and implements arbitration for a session's controlling terminal once it has been allocated.
- The controlling terminal itself is accessed by opening the special character driver associated with the name `/dev/tty` in the file system name space.

These subsystems access each other's data frequently. While this is not a problem in monolithic SVR4, it causes trouble in our implementation. Process management data structures are located in the PM, STREAMS data structures are located in the STM and the driver for `/dev/tty` is most naturally located in the OM. Thus, direct access of one subsystem's data by another is not possible in our operating system; messages must be exchanged between actors instead. Since we divided the SVR4 code for the various subsystems into separate actors, we were forced to invent a new mechanism to implement controlling terminals. We also wanted to have a mechanism that could be used in a distributed environment, and that would perform reasonably well.

In our operating system, controlling terminals are implemented using a **controlling terminal port group,** which is a port group containing the port of the STM that currently contains the controlling terminal. There is one such port group per session. The port group's UI is created via the *uiBuild*(K) Nucleus call from the session ID of the session that allocates the controlling terminal. When this allocation takes place, the STM that manages the terminal inserts its port into the controlling terminal port group for the session. The `/dev/tty` driver can then access the controlling terminal for a session by converting the session ID of the calling process into the controlling terminal port group UI, and sending a message to the STM whose port is contained in that group. Access to a controlling terminal can be terminated by removing the STM's port from the controlling terminal port group, so that further attempts at communication fail for lack of a reachable destination port.

5.5 Memory Allocation

SVR4 introduced dynamic memory allocation in the kernel via the *kmem_alloc*(D3DK) DDI function. This memory allocator takes large, virtually contiguous chunks of memory and fragments them into sizes useful to the various subsystems of the kernel. Since the SVR4 kernel is monolithic, there is only one such

---

3. CHORUS is considering removing the finite message size constraint from their message interface. This would eliminate the need for the *vmCopy*(K).

allocator in the system. In our operating system, however, we have multiple system actors, each (conceptually) with its own address space. The Nucleus has a memory allocator built into it (*rgnAllocate*(K)), but that allocator operates at the page level of granularity. Since the memory overhead caused by internal fragmentation would be prohibitive using this allocator, we decided to implement *kmem_alloc*(D3DK) in the actors, and use *rgnAllocate*(K) to allocate the large chunks of virtually contiguous memory necessary for *kmem_alloc*(D3DK) to function.

Although we were initially compelled to use a separate memory allocator for each actor because of the lack of a suitable global allocator in the Nucleus, this decision had benefits that we did not fully appreciate until we began to debug our operating system. Memory leaks (buffers allocated but never freed) were easier to localize than they would have been in monolithic SVR4, since we had information about memory allocation on a per-actor basis. Corruption of dynamically allocated memory was also easier to localize, since either a driver or other code in the actor that saw the corruption was almost always responsible for the problem.

One might wonder what the cost of having a separate memory allocator for each actor is. In our operating system, the size of the implementation of *kmem_alloc*(D3DK) and its cohorts, *kmem_zalloc*(D3DK) and *kmem_free*(D3DK), is about 15K of text, 5K of data, 2K of bss and 12K of data structures dynamically allocated during initialization, for a grand total of about 34K per actor. Considering the amount of memory required to run UNIX on most machines, we think that the cost of multiple memory allocators is reasonable and is outweighed by the benefits they provide.

5.6 Experiences with Process Management

Several race conditions were exposed by the multithreaded implementation of the PM. For example, care was required when a parent and child call *exit*(2) simultaneously, to prevent a zombie child from remaining in the system indefinitely.

In SVR4 a process hierarchy is maintained with the use of linked lists and pointers. In our implementation, an exiting parent process communicates with its child process, and vice versa, via messages. If multiple processes in the heirarchy are exiting simultaneously, it is hard to notify successive generations about their next-of-kin.

5.7 Experiences with /proc

The original SVR4 design and implementation of the `/proc` file system was constructed using a monolithic kernel implementation. The process data structures (proc structure and u area) were accessible directly from the virtual file system code. In our implementation, the process class structures are contained within the PM and the virtual file system structures are contained within the OM and therefore are not shared as in traditional monolithic UNIX implementations.

This led to a division of the `/proc` code between the two actors, with the OM handling file system operations and passing what it cannot handle on to the PM. Because of the multithreaded nature of our implementation, the guarantees and assurances provided by a single-threaded monolithic UNIX implementation taken advantage of by the existing `/proc` code could not be provided. This caused some interesting synchronization problems between the target processes and the debugging process.

Requests handled by the PM to control a target process were delivered to the PM request port. This port is serviced by several threads that execute asynchronously (and preemptively) to other PM threads and user threads. This required the use of some thread state and a pair of semaphores (with mutexes for protection) to coordinate between the target thread and the PM service thread performing some operation on the target thread.

Another interesting semantic of the `/proc` file system is the ability for a process to exit while debuggers are still accessing the process. Since the open file capability used for files in the `/proc` file system

contains the process id of the target process, that PID can be used as a search key on the PM process directory to ensure that once a process exits, successive file system calls targeting that process return an error indication to the caller. A side-effect of the search operation if the process exists is to obtain a lock on the process that prevents exit and signal operations from completing until the `/proc` operation is complete.

5.8 History Object Garbage Collection

The CHORUS paged virtual memory model uses **history objects** to defer data copies until they are required [Abrossimov 89]. In the event that a process forks, the child's data segment becomes a leaf of the binary tree rooted at the parent's data segment. After a cache miss in the child, pages are found by searching upward toward the root of the tree. As pages are modified at the root of the tree, copies are moved down to the history object.

As noted in the Abrossimov paper, if a process repeatedly forks and exits, it can leave a growing chain of history objects. We hoped not to see a problem here with "real world" applications. As it turns out, at least some large multi-user benchmark programs exhibit exactly the described behavior. Hence, garbage collection of history objects was added to the paged virtual memory model.

5.9 File System Porting Issues

A great deal of time and effort went into designing and implementing the interfaces that support the client-server model. The monolithic SVR4 VM/file system interface is an elaborate arrangement of coroutines, recursive locks and special cases (e.g., the pageout daemon). By contrast, the microkernel/external mapper model provides a small number of well-defined interfaces for the movement of data between the Nucleus and the file systems.

Our current implementation eliminates some vnode operations and adds some new ones. Those eliminated as a result of our Nucleus interface changes are *VOP_GETPAGE*(), *VOP_PUTPAGE*() and *VOP_MAP*(). Operations added are *VOP_PULLIN*(), *VOP_PUSHOUT*() and *VOP_GETACCESS*(). These interfaces support the external mapper services described in section 2.1.

We added one vnode operation for the purpose of attribute synchronization: *VOP_OBJSYNC*(). It is used to adjust time and size information in the file system specific inode. This operation may also free any allocated, but not used (section 2.2), disk space.

To support attribute distribution, tokens must occasionally be recalled from clients before an operation can proceed. An example is file truncation. We have implemented generic token services that are flexible and usable by any of the file system types.

The *rdwri*() code is never exercised for regular files, which go through the PULLIN, PUSHOUT interfaces. For directories and symbolic links, we use the buffer cache. Therefore, we were able to eliminate all knowledge of Nucleus and address space data structures from the file systems.

We have ported the **s5**, **ufs** and **nfs** file systems from the SVR4 distribution to this environment. Most functions from the porting base either survive intact or are removed completely. The new functions that must be added and other specific changes are very similar between file systems. At this time, porting a new disk based file system to this interface should be little more than a "cookbook" effort.

5.10 DDI Compliance of SVVS

Since USL introduced the DDI as the standard interface between device drivers and the kernel in SVR4, and since we had no existing non-DDI compliant drivers for our architecture, we saw no reason to to support non-compliant drivers. We did not anticipate, however, that USL would ship a version of SVVS for SVR4 that incorporated non-DDI compliant STREAMS drivers! We (eventually) got a waiver for the sections of SVVS that required the use of these drivers, so that our operating system could be certified as

being SVID 3 compliant.

Primitive Instruction Counts	
Operation	*Instruction Count*
ipcCall[call]	200
ipcCall[reply]	100
User trap to first PM instruction	77
Return to user from PM	123
Interrupt to first OM/STM instruction	151
mutexGet, no contention	5
mutexRel, no contention	9
SemV, no waiters	102
SemV, waiters, no switch	293
SemV, context switch	612
SemP, no contention	97
SemP, context switch	612

Table 1.

6. Performance

The ideal performance comparison would, of course, be to run identical benchmarks on identical hardware, varying only the operating system. We do not have that luxury: we do not have a standard implementation of SVR4 running on the S/8400. However, we do have some performance measurements.

6.1 Micro Measurements

We have instrumented our hardware to enable us to collect instruction and address traces of arbitrary length while the system is running. The measurement technique has some drawbacks: timing of external events is somewhat distorted, and the traces include all delayed branch slot instructions, whether or not they are executed. (The Motorola 88000 is a RISC processor, which always fetches the instruction immediately following a branch. The instruction may or may not be executed, depending on the branch instruction type, even if the branch is taken.) The instruction counts for various Nucleus calls are included in Table 1.

Contended mutex operations have the same cost as contended semaphore operations. The cost of traps and basic synchronization operations are comparable to those of non-microkernel operating systems. *ipcCall(K)* costs are real overhead, compared to operating systems that cannot be distributed.

At this point, little effort has been expended to optimize these costs.

6.2 Macro Measurements

Some macro benchmarking and optimization have been done on this operating system. However, the insights gained from the instruction tracing work described above have not yet been incorporated into the system. Hence, the numbers quoted here should be regarded as initial values only.

The single processor S/8400, with 32 KB of instruction cache, 32 KB of data cache, 64 MB of memory, and one disk, delivers 204 AIM III user loads.

7. Conclusion

We met the basic requirements of binary compatibility and reasonable price/performance relatively easily. System performance at this point is competitive, but not yet outstanding. Compliance with the DDI interface is complete — importing new device drivers and STREAMS modules is straightforward.

Importing new disk-based file systems is essentially a cookbook operation.

It is still too early to judge how easy it will be to track new USL releases.

The system is robust and quite stable — we have been using it in-house as our development platform now for over a year, and it has been shipping to customers for more than six months. Extending it to run on the dual processor hardware was simple: essentially no changes were required outside the Nucleus. Experimentation with distribution has begun, and the preliminary results are encouraging.

With several years of development experience using this technology, we are convinced that this is an appropriate way to build commercial operating systems.

8. Acknowledgements

Without the dedication and hard work of the following people, this project would never have seen the light of day: Francois Armand, Chris Bertin, Philip Chan, Yu Chang, Jan-Hua Chu, Gilles Courcoux, Joyce Crowley, Jean-loup Gailly, Jean-Jacques Germond, Gordon Harris, Frederic Herrmann, Billy Ho, Michael Ho, Srivatsan Kasturi, Sandy Lee, Herman Li, Sam Li, Fong-Ching Liaw, Linda Lin, Luke Lin, Jim Lipkis, Lisa Liu, Susan Lor, Jarrett Lu, David Lyle, Anup Pal, Eric Pouyoul, Ellen Reier, Mark Rooke, Marc Rozier, Bob Schatz, Mike Scheer, Graham Stott, Virendra Vase, Bhaskar Vissa, Cheng-Mu Wen, Tim Williams, Becky Wong and Kwame Yeboah.

9. References

[Abrossimov 89] V. Abrossimov, M. Rozier, M. Shapiro. "Generic Virtual Memory Management for Operating System Kernels." Proc. of 12th ACM Symposium on Operating Systems Principles, Litchfield Park, Arizona (USA), December 1989, pp. 27

[Bershad 90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, Henry M. Levy. "Lightweight Remote Procedure Call." ACM Transactions on Computer Systems, vol 8, 1, February 1990, pp. 37-55

[Cheriton 90] David R. Cheriton, Gregory R. Whitehead, Edward W. Sznyter. "Binary Emulation of UNIX using the V Kernel." Proc. of Summer 1990 USENIX Conference, Anaheim, California (USA), June 1990, pp. 73-86

[Golub 90] Davic Golub, Randall Dean, Alessandro Forin, Richard Rashid. "UNIX as an Application Program." Proc. of Summer 1990 USENIX Conference, Anaheim, California (USA), June 1990, pp. 87-96

[Guillemont 91] Marc Guillemont, Jim Lipkis, Douglas Orr, Marc Rozier. "A Second Generation Micro-Kernel Based UNIX: Lessons in Performance and Compatibility" Proc. of the Winter 1991 USENIX Conference, Dallas, Texas (USA), January 1991, pp. 13-22

[Kleiman 86] S.R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX." Proc. of the Summer 1986 USENIX Conference, Atlanta, Georgia.

[Rozier 88] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, Will Neuhauser. "CHORUS Distributed Operating Systems." Computing Systems Journal, vol 1, 4, The Usenix Association, December 1988, pp. 305-370

[Tanenbaum 90] Andrew Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, Guido van Rossum. "Experiences with the Amoeba Distributed Operating System." Communications of the ACM, volume 33, number 12, December 1990, pp. 46-63

[Williams 89] Tim Williams, "Session Management in System V Release 4." Proceedings of the Winter 1989 Usenix Conference.

Data Movement in Kernelized Systems

Randall W. Dean
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, Pennsylvania 15213

(412) 268-7654

rwd@cs.cmu.edu
Fax: (412) 681-5739

Francois Armand
CHORUS Systemes
6 Avenue G. Eiffel
78182 ST-QUENTIN-EN-Y
CEDEX-FRANCE

+33 (1) 30-64-82-00

francois@chorus.fr

This paper considers how two kernelized systems, Mach 3.0 with the BSD4.3 Single Server and CHORUS/MiX V.4, move data to and from files under a variety of circumstances. We give an overview of the kernel abstractions and system servers and describe in detail the *read()* and *write()* paths of these two systems. We then break down their *read()* and *write()* performance and compare them to two monolithic systems, Mach 2.6 MSD(BSD4.3) and System V R4.0. We then describe the compromises each of the two kernelized systems made in order to achieve a goal of performance comparable to the monolithic systems. We conclude with a description of what techniques each system uses that could benefit both each other and traditional monolithic systems.

1. Introduction

A recent trend in operating system research has been towards small kernels exporting a small set of abstractions that are used by a server or set of servers to implement the services provided by traditional operating systems [7, 9, 14, 18]. This approach to building operating system services gives OS developers a sophisticated environment for building and testing new services and meeting the needs of newer hardware platforms. A kernelized system also allows a developer to mix and match components as needed, minimizing or eliminating unneeded capabilities that are a permanent part of traditional monolithic systems. Kernelized systems have also demonstrated that they are a sound basis on which one can build distributed operating systems [2, 5] and/or provide features such as real-time [15, 19] or Object-Oriented environment [16].

For kernelized systems to gain acceptance, they must be binary compatible with and perform comparably to monolithic systems. Many papers have described efforts and experiences towards achieving these goals [6, 13]. Some authors have asserted that some services, such as, the file system, do not belong outside of the kernel [20]. One of the major fears appears to be the need for costly context switching. To meet the performance goal, a kernelized system must either make context switching faster than in systems where it is not in the critical path, or somehow avoid them entirely where possible. Data movement must also be carefully designed in these systems to avoid extra copying of data.

This paper considers how two systems, Mach 3.0 with the BSD4.3 Single Server and CHORUS/MiX V.4, achieve the performance goals by controlling data movement. We are particularly concerned with how fast these two systems can move data to and from files under a variety of circumstances. First we give an overview of the kernel abstractions and system servers and describe in detail the *read()* and *write()* paths of these two systems. We then break down their *read()* and *write()* performance and compare them to two monolithic systems, Mach 2.6 MSD(BSD4.3) and System V R4.0. Next we describe the compromises each of the two kernelized systems has made to achieve performance comparable to the monolithic systems. We

conclude with a description of what techniques each system uses that could benefit both each other and traditional monolithic systems.

2. Microkernel Abstractions

The Mach 3.0 Microkernel and the CHORUS Nucleus supply a similar set of abstractions for building systems servers [10, 17]. Unfortunately, for historical reasons, the two systems often use different names to describe the same thing. The remainder of this section describes the abstractions of Mach 3.0 and CHORUS relevant for understanding the rest of the paper using either the common name or both when necessary.

- A **Task** [4] or **Actor** [8] is an execution environment and the basic unit of resource allocation. Both include virtual memory and threads. The Mach task also includes port rights. An actor includes ports as communication resources. A task or actor can either be in kernel or user space.
- **Threads** are the basic unit of execution. A task or actor can have multiple simultaneous threads of execution. Threads may communicate via **Ports.**
- Both systems are built around Interprocess Communication or IPC.
    - Mach ports are protected communication channels [12] managed by the kernel with separate port rights residing in each task. A thread uses the local name and right for a port residing in its task to send typed data to the task having the unique receive right for that port. **Port Sets** allow a task to clump a group of ports together for the purpose of receiving from multiple ports with only one thread.
    - CHORUS IPC uses a single global name space with each port named by a Unique Identifier (UI) to send untyped data to ports. CHORUS ports, which may belong to different actors, may be grouped into **port groups**. CHORUS IPC offers the capability of broadcasting a message to all ports in a group. Actors running in supervisor space may define **message handlers** to receive messages. Instead of explicitly creating threads to receive the messages, an actor may attach a handler, a routine in its address space, to the port on which it waits for messages. When a message is delivered to the port, the handler is executed within the context of the actor using a kernel provided thread. This mechanism avoids extra copy of data and context switches when both the client and the server run on the same site. The connection of message handlers is transparent to the client of a server.
- Address spaces
    - In Mach, the address space of a task is made up of **VM Objects**. Objects often map secondary storage managed by an **External Pager** [21]. An object, which is represented by a send right to a port, must be entered into the task's address space with **vm_map** and can subsequently accessed through normal memory accesses.
    - The address space of a CHORUS actor is made up of **Regions**. Regions often map secondary storage called **segments** managed by **Mappers**. A segment is represented by a capability, a port UI and a key. CHORUS also allows segments to be read or written directly without mapping them using *sgRead()* and *sgWrite()* Nucleus calls.
- Device Access
    - Mach gives direct access to disks and other devices through *device_read()* and *device_write()*. The *device_read()* call, which, like most Mach calls, is an RPC to the kernel, returns the data in out of line memory directly from the disk driver without any unnecessary copies.
    - By contrast, the CHORUS nucleus doesn't know about any device except the clock. Instead, it allows actors to dynamically connect handlers to interrupts and traps. Device drivers are implemented this way by actors running in the supervisor address space.
- To allow for binary compatibility, Mach and CHORUS each have a mechanism for handling

Unix™ system call traps called **Trap Emulation** or **Trap Handlers.** The Mach kernel enables a task to redirect any trap number back into the user task making the trap. CHORUS has a mechanism for allowing traps to be handled by any actor runing in supervisor address space which has attached a handler to these traps.

3. System Servers

Both Mach 3.0 with the BSD4.3 Single Server and CHORUS/MiX V.4 consist of a small kernel or nucleus described in the previous section, and a server or set of servers running in user mode or possibly supervisor mode supporting Unix semantics. This section gives on overview of both of these systems server components.

3.1. Mach 3.0 with the BSD4.3 Single Server

The BSD4.3 Single Server is a single user application which exports Unix semantics to other user applications. To communicate with this server, an **Emulation Library** is loaded into the address space of all clients beginning with /etc/init and inherited subsequently by all of its children. A typical system call traps into the kernel and is redirected by the **Trap Emulation** mechanism back out into the Emulation Library. The Emulation Library sends a message to the BSD4.3 Single Server which then executes the actual Unix call and returns back through the Emulation Library to the user application. The remainder of this section describes the Emulation Library and BSD4.3 Single Server and how they support Unix files.

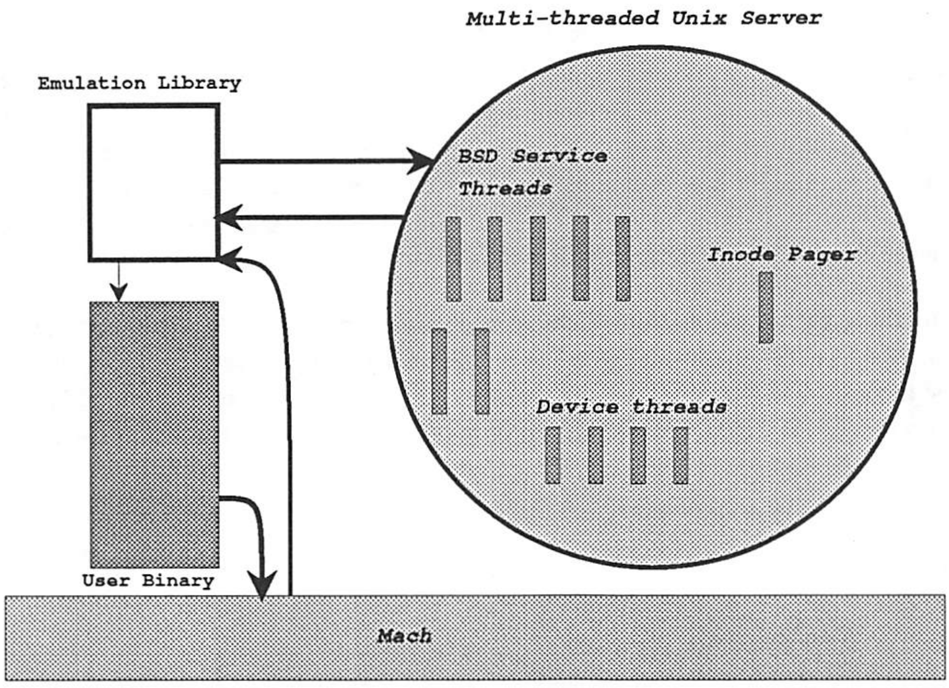


Figure 3-1: A System Call in Mach 3.0 with the BSD4.3 Single Server

3.1.1. BSD4.3 Single Server and Emulation Library Structure

The BSD4.3 Single Server is a single task made up of numerous CThreads [11]. Most of the CThreads are part of a pool which responds to messages from the Emulation Libraries of Unix processes. When they receive a message from a client Emulation Library, they internally take on the identity of that client and execute the appropriate Unix system call. Upon completion, they return the result and reenter the pool of waiting threads. Each of the remaining CThreads provides a particular service. There is the **Device Reply Thread**, which handles all responses from the server to the kernel for device interaction, the **Softclock Thread**, which implements internal timeouts and callbacks, the **Net Input Thread**, which handles network device interactions with the kernel, and the **Inode Pager Thread**, which implements an external pager backing Unix file objects.

The Emulation Library is actually a self-contained module that is loaded into the address space of BSD4.3 Single Server clients. It is entered from the kernel's Trap Emulation code in response to a system call. The Emulation Library then either handles the system call locally or forwards the request as a message to the server to handle it. To allow the Emulation Library to handle some system calls without contacting the BSD4.3 Single Server, the Emulation Library and the BSD4.3 Single Server share two pages of memory. While the Emulation Library can read both pages, it can only write one of them. The read-only page contains information that a client can already get through querying system calls such as *getpid()*, *getuid()*, and *getrlimit()*. The writeable page contains data that the client can already set through system calls such as *sigblock()* and *setsigmask()*. The writeable page also contains an array of special files descriptors used by the mapped files system.

3.1.2. Mapped Files

The BSD4.3 Single Server is capable of mapping files backed by the Inode Pager directly into the address space of clients for direct access by the Emulation Library. These mapped regions are actually windows into the Unix files that can be moved by a request from the Emulation Library. There is exactly one window of 64K bytes in size for each open file. For each mapped region, there is a corresponding file descriptor in the writeable page of shared memory. This file descriptor contains information on the current mapping window and a copy of the real file descriptor in the BSD4.3 Single Server.

To allow for Unix file semantics which permit multiple readers and writers of files and the sharing of the current file pointer, there is a **Token** scheme. The Token protects the mapping window information, the file pointer and the file size. The Token can be in three states. These are *active*, *invalid* and, to limit the number of messages sent to the BSD4.3 Single Server, *passive*. The transitions between these states is covered in section 4.

3.2. CHORUS/MiX V.4

3.2.1. The CHORUS/MiX V.4 subsystem

MiX V.4 is a CHORUS subsystem providing a Unix interface that is compatible with Unix SVR4.0. It is both BCS and ABI compliant on AT/386 and 88K platforms. It has been designed to extend Unix services to distribution such as access to remote files and remote execution.

MiX V.4 is composed of a set of cooperating servers running in independent actors on top of the CHORUS Nucleus and communicating only by means of the CHORUS IPC. The following servers are the most important:

- The **Process Manager** (PM) provides the Unix interface to processes. It implements services

for process management such as the creation and destruction of processes and the sending of signals. It manages the system context of each process that runs on its site. When the PM is not able to serve a Unix system call by itself, it calls other servers, as appropriate, using CHORUS IPC.

- The **Object Manager** (OM), also named the **File Manager** (FM), performs file management services. It manages various file system types such as S5, UFS, and NFS. It also acts as a CHORUS Mapper for "mappable" Unix files and as the Default Mapper for swap space management. Disk drivers are generally part of the Object Manager.
- The **Streams Manager** (StM) manages all stream devices such as pipes, network access, ttys, and named pipes when they are opened. It cooperates with the Object Manager which provides the secondary storage for the files' meta-data.

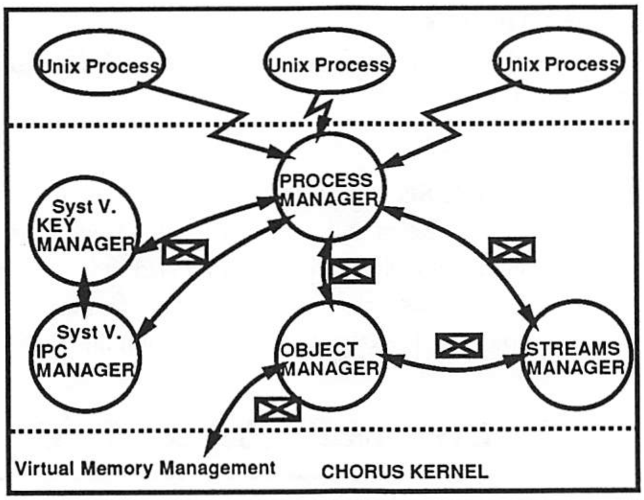


Figure 3-2: CHORUS/MiX V.4 subsystem

All MiX servers are fully independent and do not share any memory and thus can be transparently distributed on different sites. Although they may run in either user space or supervisor space they are usually loaded in the supervisor address space to avoid numerous and expensive context switches each time a new server is invoked. This paper discusses the case where all servers are loaded into supervisor space.

3.2.2. Regular File Management in CHORUS/MiX V.4

The management of regular files is split between three components in MiX V.4: the Object Manager, the Process Manager and the CHORUS Nucleus. Pages of regular files are read, written and cached using the Virtual Memory services, *sgRead()* and *sgWrite()*, without having to map them. This allows the support of mapped files and provides the benefit of the Virtual Memory mechanisms for caching files from local or remote file systems. The Virtual Memory cache replaces the buffer cache, moreover in the distributed case, file consistency are maintained by the same mechanisms that are used to provide distributed shared memory.

In MiX V.4, open files are named by a capability built from a port Unique Identifier and a 64-bit key only meaningful to the server. This capability is returned to the PM by the OM when the *open()* system call is made. All opens of the same file get the same capability. For each open system call, the PM manages an open file descriptor similar to the *file_t* structure of a native Unix, that handles flags and the current offset. In addition, the PM manages an **object descriptor** for each file in use on its site. This object descriptor handles the following information: size of the file, mandatory locks posted against the file (if any), last access and last modification time and, the capability of the file exported by the OM. The PM

uses this information to convert *read()/write()* Unix system calls into *sgRead()/sgWrite()* CHORUS Nucleus calls.

3.2.3. File Size Management

Due to the separation between the Process Manager and the Object Manager, a benefit which allows them to potentially be distributed on different nodes, the file size is protected by a **Token.** Since the Token may have been granted to another site or recalled by the OM, the PM must check whether the file size information it holds is valid or not before using the file size. If the information is not valid, the PM must retrieve the information first.

4. *Read()* and *Write()* Path Analysis

This section takes a detailed look at the *read()* and *write()* paths of both Mach 3.0 with the BSD4.3 Single Server and CHORUS/MiX V.4.

4.1. Mach 3.0 with the BSD4.3 Single Server *read()* Path

This section describes how the *read()* system call works in Mach 3.0 with the BSD4.3 Single Server. When a user makes a *read()* call the kernel redirects that call trap back into the Emulation Library. After making sure that the file is mapped, that it has a valid Token, and that the mapping is into the desired part of the file, the Emulation Library copies the data from the mapped region into the user's buffer and returns.

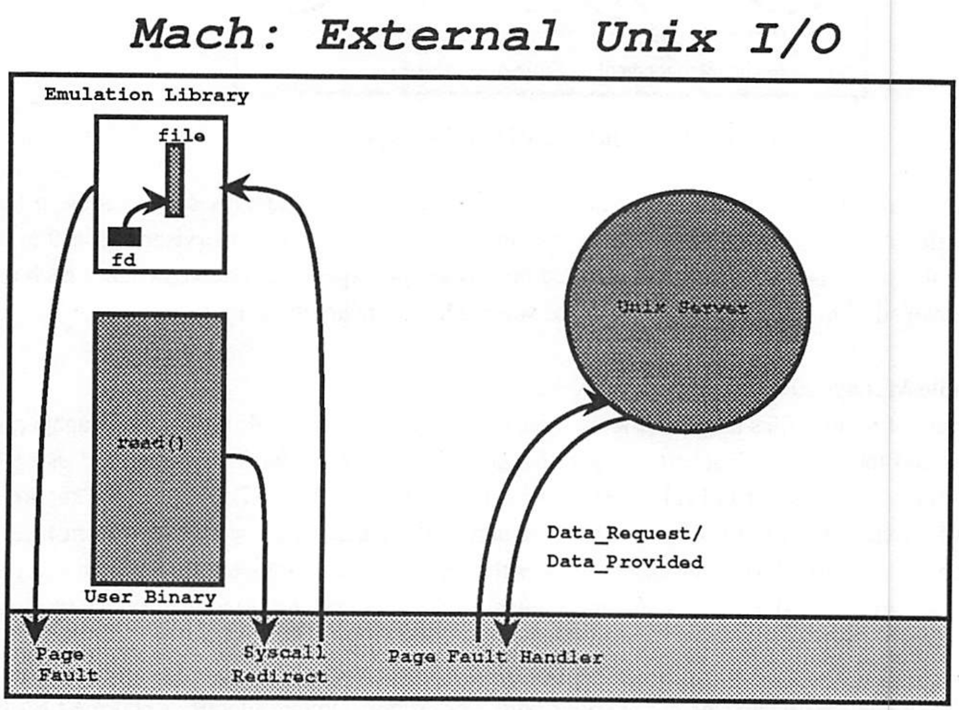


Figure 4-1: Mach 3.0 with the BSD4.3 Single Server Mapped File Read

4.1.1. *Read()* in the Emulation Library

To examine the file descriptor in the writeable shared page, the Emulation Library must first acquire the **share lock** which protects this region. A share lock is different from a simple spin lock in that the BSD4.3 Single Server does not trust the Emulation Library. The Emulation Library resides in user memory and can be over-written at any time by an incorrect or malicious user program. Therefore, the BSD4.3 Single Server must take measures to ensure it does not deadlock on this share lock. The share lock is implemented as a simple spin lock in the Emulation Library and as a test-and-set and block with timeout in the server. When the server blocks, it requests a callback from the Emulation Library when the Emulation Library releases the lock. If the callback is not received in a configurable amount of time, the server assumes the client is malicious or incorrect and kills the task.

Once the Emulation Library has acquired the share lock, it can examine the state of the Token. If the Token is *invalid*, the Emulation Library must send a message to the server to acquire the Token. If the Token is *passive*, it may be switched directly to *active* without contacting the server Once the Emulation Library has the Token, its file pointer and mapping info are guaranteed to be correct. If the mapping information is not adequate for the current call, the Emulation Library can send a message to the server to change the mapping. With a valid mapping, the Emulation Library simply copies the data into the user's buffer, updates the file pointer, and releases the Token. If the BSD4.3 Single Server has indicated, by setting a bit in the file descriptor, that another task is waiting for the Token, the Emulation Library sends a message to the server to completely release the Token. If no call-back is needed, the Emulation Library moves the Token to *passive*.

4.1.2. *Read()* in the BSD4.3 Single Server

The BSD4.3 Single Server can become involved in a *read()* call in a number of ways. The Emulation Library can call it directly in order to request a Token, request a window remapping, or release a Token. The BSD4.3 Single Server can also be called by the Mach kernel when the Emulation Library faults on mapped data and the Inode Pager must be contacted to satisfy the fault. Each of these operations is described in detail below.

A Token request is straightforward to process. The server keeps track of who has the Token and examines the state of that process's file descriptor. If the holder of the Token has it *passive*, the server invalidates the Token and grants it to the requester. If the Token is *active*, the server leaves a call back request with the holder and waits for it to send a Token release call.

A request to change the mapping window is extremely simple when just reading from a file. All the server has to do is deallocate the existing window into the file and generate a new mapping which covers the range necessary for the *read()*. The handling of the remapping operation for *write()* is covered in detail in section 4.2.

When the Emulation Library first touches a page of data mapped into its address space, a page-fault occurs. The kernel receives this fault and resolves it in one of two ways. If the page is already present in the kernel VM system, but the task does not have a valid mapping, then the kernel enters the page mapping and returns from the fault. If the page is not present in the kernel, the kernel sends a *memory_object_data_request()* message to the External Pager backing this object. In the case of Unix file data, this is the Inode Pager in the BSD4.3 Single Server. The Inode Pager then reads the appropriate data off disk with *device_read()* and provides this page back to the kernel.

For historical reasons, the Inode Pager uses the buffer cache interface to read and write from and to disk.

This leads to unnecessary caching in the BSD4.3 Single Server. This also results in the Inode Pager using a less efficient and now obsolete interface *memory_object_data_provided()* instead of *memory_object_data_supply()* to return the data to the kernel. The former requires a copy within the kernel, whereas the latter uses a page stealing optimization that eliminates the copying. In the page stealing optimization, the VM system just removes the physical page from the server and places it directly in the Memory Object without copying it.

4.2. Mach 3.0 with the BSD4.3 Single Server *write()* Path

The *write()* path is almost identical to the *read()* path. The differences lie in correctly managing the file size and writing out dirty pages in a timely fashion. As in the *read()* case, the Emulation Library sets up a valid mapping and copies the user's data into the window. A valid mapping may exist past the end of the file, so there is no need to handle file extension as a special case in the Emulation Library.

If the Emulation Library write faults on a page that is in the file, the Inode Pager returns the page from the disk file to be over written. In the case of *write()* faulting on a new page, such as filling a whole or extending the file, the Inode Pager returns *memory_object_data_unavailable()* which causes the kernel to supply a zero filled page to the client. If the *write()* extends the file, the Emulation Library updates the file size field in the local file descriptor.

Actual writes to disk and propagation of the file size occur when the Token is taken away from a writer, when the writer changes its mapping, or when the kernel needs free pages and starts sending the mapped file pages to the Inode Pager to be cleaned. In the first two cases, the mapped file code first updates the file size from the writer's file descriptor. The server then calls *memory_object_lock_request()* to force the kernel, which knows which pages have actually been written by the user, to request cleaning of the dirty pages. This generates messages from the kernel to the pager requesting that the dirty pages be written out. When the Inode Pager receives the dirty pages from the kernel, it writes them out to disk.

Disk block allocation is delayed until the data is actually written out to disk. It would be possible for the Inode Pager to allocate new blocks in a timely fashion since it gets a data request message when the Emulation Library page-faults on the new page to be written. The Inode Pager currently ignores the write fault attribute when returning the empty page to satisfy the page-fault. By not allocating at the time of the initial fault, the semantics for failure in the BSD4.3 Single Server are not the same as Mach 2.6 MSD(BSD4.3).

The difficulty occurs when there was a involuntary request to clean pages from the kernel driven by a memory shortage. In the previous cases where dirty pages and the file size were written out, the writing was initiated by a call from the Emulation Library. In this case the Emulation Library could be in the middle of a *read()* or *write()* call, so the Inode Pager must read the file size from the current holder of the Token without taking that Token away.

4.3. CHORUS/MiX V.4 *read()* Path

This section illustrates how a Unix *read()* system call is handled in the CHORUS/MiX V.4 system. When a Unix process traps to the system, it executes the trap handler connected by the Process Manager. The PM performs various checks and invokes the CHORUS Nucleus call *sgRead()*. The Nucleus looks for the corresponding page. If the page is found, data is directly copied into the user's buffer. If the page is not found, an upcall to the appropriate mapper is performed by sending a message to the port whose UI is part of the file capability. In this case, the mapper, which implements the disk driver, reads the data from the

disk and replys to the Nucleus which copies out the data into the user's buffer. The overall mechanism is summarized in the figure 4-2.

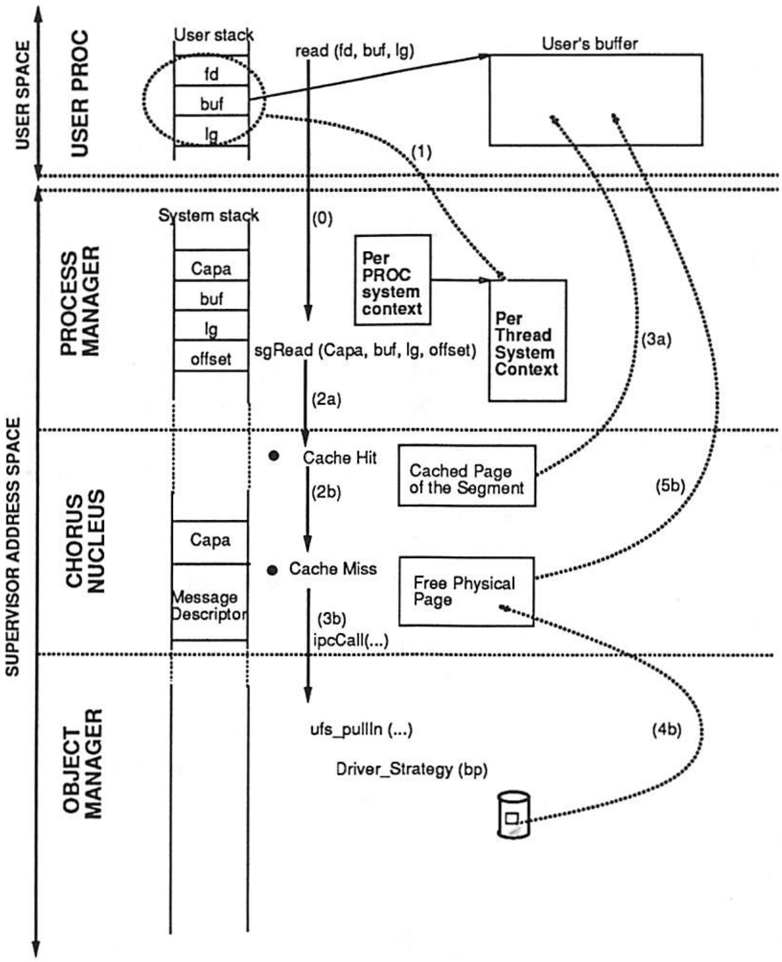


Figure 4-2: CHORUS/MiX V.4 read path

4.3.1. Process Manager

As in any Unix system, the library part of a system call runs in user space and builds a stack frame with the system call arguments and then traps. The trap is handled by the CHORUS Nucleus which redirects the invocation to the trap handler connected by the PM at init time by means of the *svTrapConnect()* Nucleus call (arrow 0 in figure 4-2).

The trap handler executes in supervisor space in the context of the process invoking the system call. The system stack used by the trap handler is the one provided by the CHORUS Nucleus at thread creation time. Using software registers provided by the CHORUS Nucleus, the handler retrieves the thread and process system context. The user stack frame is then copied into the thread structure since multiple threads can run concurrently within a process (arrow 1).

Once the file descriptor has been validated, the PM distinguishes between files that are accessible through the Virtual Memory Interface, such as regular files and mappable character devices, and other files such as directories and streams that are read and written by sending messages to the appropriate server. The PM then acquires the current offset for the file. This requires that it hold a Token in the distributed

release of the system. When the system runs on a single site, the PM is assured that the offset information it has is always valid.

The PM then must determine whether the read starts beyond the current End Of File, or not, and thus must acquire the file size information. This information is protected by a token mechanism which is separate from the Token for the current offset. At the time the *read()* occurs, the Token might not be present in the PM, but may have been recalled by the OM. Finally, after having checked the read operation against mandatory locks (if any), the PM invokes the *sgRead()* CHORUS Nucleus call.

4.3.2. CHORUS Nucleus

The CHORUS Nucleus first tries to retrieve the local cache [1] associated with the segment being accessed. A local cache is created for the segment upon the first access to the segment. Through the local cache descriptor the Nucleus accesses the pages of the segment that are present in main memory, and their associated access rights whether readable or writable. In case the page being read is present (arrow 2a) in the local cache, data is copied out to the user's buffer (arrow 3a). If not present, a free page is acquired (arrow 2b) and the Mapper managing the segment is invoked (arrow 3b). Upon return from the Mapper, the data is copied out into the user's buffer (arrow 5b).

It should be noted that the *sgRead()* operation insures the seriability of concurrent overlapping reads and and writes on a segment.

4.3.3. Mapper

In MiX V.4, regular files are managed by the Object Manager, which acts as the Mapper for these files and thus handles pullIn (read page) and pushOut (write page) requests for these files. When the Object Manager runs in supervisor address space it uses a message handler to handle incoming requests. The client thread is made to run in the server context. Direct access to invoker message descriptors is permitted thus avoiding extra copy of data.

The Object Manager then invokes the corresponding function (e.g.: *ufs_pullin()*) using an extension to the vnode_ops mechanism. The file system specific pullin function converts the page offset in the file into a physical block number and then invokes the disk driver to load data. The buffer area is set up so that it points directly to the physical page provided by the Nucleus and passed as part of the reply message (arrow 4b). The OM only buffers inodes, directory and indirect blocks.

4.4. CHORUS/MiX V.4 *write()* Path

The basic mechanisms are similar to the ones described for the *read()* system call. However, a *write()* may extend a file or fill a hole in the file. In such a case, the system must immediately allocate disk blocks for the written data.

When the Object Manager returns a page of a file to the Nucleus, it returns also an associated access right enabling the page to be read-only or read-and-written. When a process wants to extend a file by writing after the EOF, the Nucleus requests the entire page for read/write access if there was a cache miss, or only the write permission if there was a cache hit, with no previous write access granted by the Mapper.

Before the Object Manager returns a write permission for a page, it insures that blocks will be available later to write the data on the disk. Blocks are allocated to the file at the time the write permission is required. When the file is closed, blocks that have been allocated but which are not used (past the EOF) are

returned to the free block pool.

Finally if the extension of the file has been successful, the PM changes the size of the file. This requires that the PM has the file size token with a write access granted to the size information.

5. Performance

This section contains the performance breakdown for the *read()* and *write()* calls described in the previous sections plus composite numbers for both these systems and from Mach 2.6 MSD(BSD4.3) and System V Release 4.0/386. The micro-benchmarks show where the time is consumed in the full *read()* and *write()* call. The benchmarks for Mach 3.0 with the BSD4.3 Single Server and Mach 2.6 MSD(BSD4.3) were run on a HP Vectra 25C 80386/25Mhz with 16MB and 32K cache, a WD 1007 disk controller and a 340MB disk. The benchmarks for CHORUS/MiX V.4 and System V R4.0 were run on Compaq Deskpro 80386/25Mhz with 16MB of memory and 32K SRAM cache, a WD 1007A disk controller and a 110MB disk drive. The testing methodology was to run a given test three to ten times and to report the median value.

As a number of tests show, the two test platforms, HP Vectra and Compaq Deskpro, have quite different disk and memory throughput performance characteristics. Basically, the HP Vectra memory throughput is between 40% and 50% higher than the Compaq Deskpro throughput (see table 5-1). In comparing the performance of the different systems below, it is essential to remember these differences and relate the figures to the maximum memory and disk throughput of the two different platforms.

To measure the performance of the memory system a simple bcopy was run on both machines. Repeated movement of 4096 bytes was used for the cached result and repeated movement of between 256k and 1M was used for the uncached result. Table 5-1 shows throughput and the time for copying a page of 4096 bytes.

	HP Vectra	Compaq
Cached	21.2MB/sec	14.7MB/sec
	184μs	265μs
UnCached	10.4MB/sec	6.4MB/sec
	379μs	615μs

Table 5-1: Bcopy of 4096 bytes

Two of the primary primitives used by the kernelized systems are trap times and Remote Procedure Call or RPC times. We measured the cost of reaching the Emulation Library or Trap Handler in the kernelized system, versus the cost of a kernel trap in the monolithic systems. For this test we used *getpid()* on all architectures. To show how much overhead there is in reaching the Emulation Library or Trap Handler for a Unix call, including such things as checking for signals, we measured the cost of a null trap into the kernel. Table 5-2 also shows RPC times for the two kernelized systems. The Mach 3.0 RPC times are between two user tasks since this is the type of communication that occurs between the Emulation Library and the BSD4.3 Single Server. The Chorus Null RPC times corresponds to a null *ipcCall()* issued from a supervisor actor to another supervisor actor handling messages via a message handler, which is what is used between PM and OM, and between Chorus Nucleus and OM. CHORUS/MiX V.4 has a more expensive getpid() call than the native SVR4 implementation. This is due, at least partially, to the MiX PM having already implemented support for multithreaded processes and multiprocessor platforms. Thus, MiX

has some synchronization mechanisms that are not present within the native Unix SVR4.

	HP Vectra		Compaq	
	Mach 3.0	Mach 2.6	CHORUS/MiX V.4	SVR4.0
Null Trap	40µs	61µs	36µs	85µs
Trap to Unix	61µs	61µs	119µs	85µs
Null RPC	310µs		83µs	

Table 5-2: Trap and RPC times

The primary measurement we are concerned with in this paper is *read()* and *write()* throughput. Table 5-3 measures throughput on all four systems for moving data to and from disk plus the extrapolated time it took for each 4096 byte page. The *read()*'s and *write()*'s were large enough so no caching effects would be seen. As can be seen, the only case where the kernelized system does not perform as well as the monolithic system is the Mach 3.0 *write()* which performs 6% slower.

	HP Vectra		Compaq	
	Mach 3.0	Mach 2.6	CHORUS/MiX V.4	SVR4.0
Read	320KB/sec	320KB/sec	270KB/sec	270KB/sec
	12.5ms	12.5ms	14.8ms	14.8ms
Write	300KB/sec	320KB/sec	250KB/sec	250KB/sec
	13.3ms	12.5ms	16.0ms	16.0ms

Table 5-3: Read and Write Throughput

To approximate the cost of the code path for *read()* and *write()* we measured the cost of *read()*'s without requiring disk access. For this test we did a sequential read of a file of approximately 2 megabytes in size for the kernelized systems and SVR4 and a file as large as possible and still able to fit into the buffer cache for Mach 2.6. Table 5-4 shows these results.

HP Vectra		Compaq	
Mach 3.0	Mach 2.6	CHORUS/MiX V.4	SVR4.0
4.3M/sec	4.3M/sec	3.3M/sec	3.0M/sec
900µs	900µs	1100µs	1300µs

Table 5-4: Cached Read Throughput

The next benchmark measures the cost for repeated calls to *read()* or *write()* of one byte with a *lseek()* between each iteration. So that the *lseek()* time can be removed for further comparisons, a benchmark to measure *lseek()* time is included. Table 5-5 shows the results of these tests. The read time for CHORUS/MiX V.4 is consistent with the equivalent time measured for SVR4: the difference (80µs)corresponds roughly to the additional cost of the trap handling (34µs for the seek call and for the read call). Among the four systems, all of them except SVR4 exhibit equivalent times for reading and writing one byte. The reason why SVR4 writes much faster one byte than it reads it, is quite unclear.

Table 5-4 measured the cost of reading 4096 bytes from a file when reading sequentially through the entire file. The difference between that result and the results shown in table 5-5 should be the difference of

	HP Vectra		Compaq	
	Mach 3.0	Mach 2.6	CHORUS/MiX V.4	SVR4.0
Lseek	95µs	79µs	155µs	121µs
Read	210µs	390µs	710µs	630µs
Write	200µs	380µs	720µs	420µs

Table 5-5: Cached Read and Write Times

an uncached and a cached bcopy and the addition of a fast page-fault resolved in the VM cache. For the Mach 3.0 system table 5-4 also includes the cost of a mapping window change every 16 iterations of 4096 bytes. To account for the mapping window change operation in Mach 3.0 with the BSD4.3 Single Server, a test was run which *lseek()*ed 61440 bytes after each 4096 byte read to force a mapping operation on each *read()*. This resulted in a cost of 2250υs for each iteration including the *lseek()* cost. By looking at 16 reads covering 64KB, we can extrapolate the cost of just the mapping operation and the cost of a page-fault. The measured result from table 5-4 is 900µs times 16 which must be equal to 2250µs plus 15 times the quantity 115µs, from table 5-5, plus 379υσ plus the page-fault time. From this, the cost of the fast page-fault time is 316µs. By breaking down the 2250µs measurement, we get the cost of the mapping operation to be 1440µs.

Another test of interest both for understanding the break down of *read()* costs and for comparison between the kernelized systems and monolithic systems is the reading of data directly from raw disk partitions. Table 5-6 shows the throughput and per 4096k byte page performance of Mach 3.0 and Mach 2.6 for different size reads. When these tests were run with block increments of one between each read, the performance was lower than that actually measured for a full *read()*. This result is consistent with the BSD4.3 Single Server's uses of readahead which will not consistently miss disk rotations like this test must have. To account for this and better simulate what is happening in the BSD4.3 Single Server, various block increments from one to eight times the data size were used in the benchmark and the increment which produced the best result was reported. An interesting point about the results is Mach 2.6 reads from raw partitions slower than it reads from Unix files.

Bytes	4KB	8KB	16KB	32KB	64KB
Mach 3.0	10.5ms	10.0ms	7.5ms	7.5ms	6.5ms
device_read()	380KB/sec	410KB/sec	530KB/sec	530KB/sec	620KB/sec
Mach 2.6	14.3ms	14.3ms	14.3ms	14.3ms	14.3ms
read()	280KB/sec	280KB/sec	280KB/sec	280KB/sec	280KB/sec

Table 5-6: Raw Read Performance

The CHORUS Nucleus does not provide any direct access to devices. MiX Object Manager accesses directly the disk driver as a native Unix implementation does, through a bdevsw/cdevsw interface. Tests were done to measure the disk throughput from within the OM using 4KB, 8KB 16KB and 32KB transfers. The test was run in two different ways: the first run was reading the disk sequentially while the second run was always reading the same block of the disk. As the controller as an internal cache reading the same block goes faster than reading the disk sequentially.

As it was difficult to get the same measure for Unix SVR4, we measured the disk throughput through the raw disk interface using standard read system calls, the throughput achieved is 530KB/sec for 4096 byte

Bytes	4KB	8KB	16KB	32KB
Sequential	7.1ms	7.1ms	7.1ms	7.1ms
read	560KB/sec	560KB/sec	560KB/sec	560KB/sec
Same Block	4.49ms	4.19ms	4.10ms	4.00ms
read	890KB/sec	952KB/sec	975KB/sec	990KB/sec

Table 5-7: CHORUS/MiX V.4 Raw Read Performance

pages.

The results in table 5-8 were gathered to show the combined effect of reading from disk and supplying the data to the VM system and to show what potential improvements could be made by tuning the file system implementation to the kernelized architecture. The Data_Provided column corresponds to *memory_object_data_provided()* the obsolete call that is used in the current implementation. Data_Supply corresponds to *memory_object_data_supply()*, the new Mach call which has the previously mentioned page stealing optimization. Like the results from table 5-6, the block increment was adjusted to remove disk rotation misses. Both tests reached maximum throughput of 530KB/sec at 16k reads and maintained that speed for larger block sizes.

Comparison of Mach Per Page Fault Cost			
Bytes	4KB	8KB	16KB
Data_Provided	12.1ms	10.0ms	7.5ms
	330KB/sec	400KB/sec	530KB/sec
Data_Supply	11.5ms	10.0ms	7.5ms
	346KB/sec	410KB/sec	530KB/sec

Table 5-8: Mach 3.0 Data_Provided and Data_Supply

Measures have been taken within the CHORUS Nucleus to measure the cost of a pullIn operation: a loop for loading the same from the Mapper in physical memory has been measured: this includes the RPC from the CHORUS Nucleus to the Mapper (83µs). This has been run with a 4096 byte page size. The time reported is 5ms which leads to a throughput of 800KB/sec. As this test doesn't really access the disk but only the cache of the controller, this figures must be compared to the figures achieved by reading continually the same block of 4096 bytes from the raw disk (4.49 ms, and 890KB/sec). The overhead introduced by the Object Manager for loading page can then be deduced as 5ms - 4.49 ms: 0.51 ms. These 510µs comprise the 83µs due to the RPC from the Nucleus to the OM. Thus, the actual overhead induced by the OM remains to some 430µs.

To bring together all of the previous results, table 5-9 shows a break down of the *read()* path with associated costs. The micro-benchmark total comes out only 4% higher than the measured result. Since the measurements for *device_read()* and *memory_object_data_provided()* are only an approximation, the measured and projected totals can be considered the same. The Mach 2.6 numbers were included for comparison. Since there is no easy way to measure the internal access time to the disk driver for reading, an extrapolated value was supplied for read which yielded a identical measured and projected total.

While the end measured result for *read()* and *write()* performance is the same in Mach 3.0 with the BSD4.3 Single Server and Mach 2.6 MSD(BSD4.3), table 5-9 shows a result which may question

Operation	Time (μs)	Time (μs)
	Mach 3.0	Mach 2.6
Trap to Unix	61	61
Misc Costs	54	250
1/16th Remap Window	90	N/A
Pagefault	316	N/A
Read from Disk	10500	11810
Data Provided	1600	N/A
Bcopy Uncached to User	379	379
Total w/o Disk Read	2500	690
Total	13000	12500
Measured Total	12500	12500

Table 5-9: Mach 3.0 with the BSD4.3 Single Server *Read()* Path

scalability the result under load. Because the disk latency is so high, the 262% increase in processing overhead necessary in the BSD4.3 Single Server is hidden. Further measurements should look at whether the effects of this increased processing outweigh the benefits seen in cached performance where the BSD4.3 Single Server *read()* overhead is only 54% of the Mach 2.6 MSD(BSD4.3) *read()* overhead.

6. Conclusions

6.1. Compromises

The kernelized systems have achieved their performance in a number of cases by compromising some of their modularity and portability goals.

Mach 3.0 with the BSD4.3 Single Server makes liberal use of the shared memory pages between the BSD4.3 Single Server and Emulation Library. While this works well on a uniprocessor or shared memory multiprocessor, it would not work well on a NUMA or NORMA machine. The mapping information which is used by the Emulation Library could be updated by messages instead of being written directly by the BSD4.3 Single Server. This would eliminate the *passive* Token state optimization since there is no mechanism for an upcall from the BSD4.3 Single Server to the Emulation Library. A dedicated thread in the Emulation Library for upcalls would solve this problem but with the cost of adding an additional thread creation at each *fork()* call. Another solution would be to have a proxy task running on each node on behalf of the BSD4.3 Single Server to share the memory with the Emulation Library and respond to upcalls. The cost here would be at startup in the creation of an additional task per node.

CHORUS/MiX V.4 is capable of runnings its system servers in either supervisor or user mode. The configuration used for the previous sections measurements was the version where the servers reside in supervisor mode. This results in no context switches and faster IPC times to read or write from/to the disk. Moreover, CHORUS/MiX V.4 servers do not share any memory, thus it is quite easy to make a component evolve without changing the other servers as long as the protocol between them remains unchanged. The configuration where CHORUS/MiX V.4 is running with its servers in user mode is primarily intended to be used for debugging purposes, thus no particular attention has been paid yet to achieve similar performances

in such a configuration. However some experiments have been done in the previous release of MiX (MiX V3.2) to measure the additional costs that such a configuration will imply. Further details can be found in [3].

One of the key point in the design of CHORUS/MiX V.4 has been the introduction of the message handler mechanism. However, this mechanism should be extended to work whether the receiver of a message is running in supervisor address space or user address space.

6.2. Cross Pollenation

A number of techniques used by CHORUS/MiX V.4 could readily be adopted by Mach 3.0 with the BSD4.3 Single Server. The movement of device drivers such as disk drivers out of the kernel or nucleus into systems servers is one good example of this. By locating the device manipulation code in the same task which needs the data, context switches and data copies can be avoided. For this to really work on a Mach based system, the device driver needs to really be able to run in user mode where systems servers are run. This is feasible on architectures which allow user mode access to device registers such as the R2000/R3000 based DECstation and 80386/80486 based platforms. The current version of the BSD4.3 Single Server for the DECstation platform uses a user mode ethernet driver and preliminary work has been done on moving disk drivers.

Another technique which CHORUS/MiX V.4 uses which could be adopted by Mach 3.0 with the BSD4.3 Single Server is the concept of a handler. By having the kernel redirect the trap directly into another address space, CHORUS avoids the protection problem the BSD4.3 Single Server has with the Emulation Library and avoids the associated RPC needed by the Emulation Library to cross into a trusted server. The question about this technique, is whether the additional cost of always redirecting into another address space outweighs the cost of the occasional RPC. Since Mach 3.0 servers are run in user mode, the cost of this redirection may turn out to be much higher than that seen by CHORUS for the redirection into a supervisor mode actor.

Mach 2.6 MSD(BSD4.3) could benefit from a mapped file system using the Emulation Library technique. This will only work because it already has the complex VM system, a fast IPC, and the Trap Emulation mechanism that resides in Mach 3.0. In general, monolithic systems can not benefit the kernelized techniques because they do not have the necessary mechanisms for building system servers with new structures.

To fully compare the two approaches taken by Mach 3.0 and CHORUS, one could use the CHORUS Trap Handler mechanism to implement an Emulation Library instead of a Process Manager in CHORUS. As long as device drivers are part of CHORUS subsystems' specific actors, it will be difficult to have multiple subsystems running simultaneously sharing the same devices. The isolation of the drivers out of the servers as done in Mach would help to solve this issue. This has been experimented in CHORUS/MiX V3.2 and described in [3].

6.3. Final Words

Section 5 shows that Mach 3.0 with the BSD4.3 Single Server and CHORUS/MiX V.4 have achieved performance in the movement of data comparable to the monolithic systems which they are compatible with. In no case was *read()* or *write()* throughput less then 94% of the performance of the monolithic system. Many of the micro-benchmarks clearly indicated better performance on kernelized systems for certain types of cached access.

There is also still clear room for improvement in Mach 3.0 with the BSD4.3 Single Server. By moving to the *memory_object_data_supply()* call from the obsolete interface and better optimizing read sizes and readahead for the performance of the microkernel, disk throughput could approach 600KB/sec or almost a 100% improvement over the existing BSD4.3 Single Server and Mach 2.6 MSD(BSD4.3) systems.

CHORUS/MiX V.4 may also expect some significant improvement since no readahead is used in the systems that have been described and measured. Pages being pushed out to the disk are written synchronously, adding some asynchronicity should help to handle multiple disk access more gracefully by making the disk sort algorithm much more useful.

Regarding the history of microkernels, their current performance has been achieved through multiple iterations which allows them to now be used in commercial products as opposed to just being interesting research tools. Yet, microkernel based systems are still very young and have not benefited from the thousands of man-years that have been spent to make monolithic systems as fast as they are now. Even as young as they are, we believe that kernelized systems have shown themselves to be both flexible and fast enough for the challenging task of building file systems and moving data. You can only imagine what kernelized systems will look like a few years from now, after receiving a fraction of the effort that has gone into monolithic systems in the past.

7. Availability

Mach 3.0 is free and available for anonymous FTP from cs.cmu.edu. The BSD4.3 Single Server is available free to anyone with an AT&T source license. Contact mach@cs.cmu.edu for information. CHORUS is available from CHORUS Systemes. Contact info@chorus.fr for licensing information. Reports and documentation is freely accessible by anonymous FTP from opera.chorus.fr.

8. Acknowledgements

We would like to thank those who have read and commented on this paper throughout its development. In particular, we thank Dr. J. Karohl, Brian Bershad, Jim Lipkis, Michel Gien, Marc Rozier, Brian Zill and especially Peter Stout.

References

[1] Abrossimov V., Rozier M., Shapiro M.
Generic Virtual Memory Management for Operating System Kernels.
In *Proceedings of the Twelvth ACM Symposium on Operating Systems Principles*. December, 1989.

[2] Armand F., Gien M., Herrmann F., Rozier M.
Revolution 89: or Distributing Unix brings it back to its Original Virtue.
In *Proceedings WEDMS I*. October, 1989.

[3] Francois Armand.
Give a Process Manager to your drivers!
In *Proceedings of EurOpen Autumn 1991*. September, 1991.

[4] Baron, R.V. et al.
MACH Kernel Interface Manual.
Technical Report, School of Computer Science, Carnegie Mellon University, September, 1988.

[5] Joseph S. Barrera III.
Kernel Support for Distrubted Memory Multiprocessors.
PhD thesis, School of Computer Science, Carnegie Mellon University, To be published, 1992.

[6] Bricker, A., Gien, M., Guillemont, M., Lipkis, J., Orr, D., Rozier, M.
A New Look at Microkernel-Based UNIX Operating Systems: Lessons in Performance and Compatibility.
In *Proceedings of EurOpen Spring 1991 Conference*. May, 1991.

[7] Cheriton, D.R., Whitehead, G.R., Szynter, E.W.
Binary Emulation of UNIX using the V Kernel.
In *Proceedings of Summer 1990 USENIX Conference*. June, 1990.

[8] Chorus Systemes.
CHORUS Kernel v3 r4.0 Programmer's Reference Manual.
Technical Report CS/TR-91-71, Chorus Systemes, September, 1991.

[9] Rozier, M., et. al.
CHORUS Distrbuted Operating Systems.
Computing Systems 1(4), December, 1988.

[10] Chorus Systemes.
CHORUS Kernel v3 r4.0 Specification and Interface.
Technical Report CS/TR-91-69, Chorus Systemes, September, 1991.

[11] Eric C. Cooper and Richard P. Draves.
C Threads.
Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, February, 1988.

[12] Draves, R.P.
A Revised IPC Interface.
In *Proceedings of the USENIX Mach Workshop*, pages 101-121. October, 1990.

[13] Draves, R.P., Bershad, B., Rashid, R.F., Dean, R.W.
Using Continuations to Implement Thread Management and Communication in Operating Systems.
In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*. April, 1991.

[14] Golub, D., Dean, R.W., Forin, A., Rashid, R.F.
Unix as an Application Program.
In *Proceedings of Summer 1990 USENIX Conference*. June, 1990.

[15] Guillemont M.
Real Time in a Distributed Computing Environment.
Computer Technology Review , October, 1990.

[16] Habert, S., Mosseri, L., Abrossimov, V.
COOL:Kernel support for object oriented environments.
In *Proceedings of OOPSLA'90 - Canada Ottawa*. 1990.

[17] Keith Loepere.
MACH 3 Kernel Principles.
Open Software Foundation and Carnegie Mellon University, 1992.

[18] A. S. Tannenbaum and S. J. Mullender.
An Overview of the Amoeba Distributed Operating System.
CWI Syllabus 9.
1982

[19] Tokuda, H., Nakajima, T., Rao, P.
Real-Time Mach: Towards a Predictable Real-Time System.
In *Proceedings of the First Mach USENIX Workshop*, pages 73--82. October, 1990.

[20] Brent Welch.
The File System Belongs in the Kernel.
In *Proceedings of the Second USENIX Mach Symposium*. November, 1991.

[21] Michael Wayne Young.
Exporting a User Interface to Memory Management from a Communication-Oriented Operating System.
PhD thesis, School of Computer Science, Carnegie Mellon University, November, 1989.

Distributed Abstractions, Lightweight References

Marc Shapiro
Mesaac Makpangou
INRIA, B.P. 105, 78153 Le Chesnay Cédex, France
tel.: +33 (1) 39-63-53-25
marc.shapiro@inria.fr

March 1992

1 Introduction

The SOR group at INRIA has been researching OS support for objects [2, 9]. This research has identified a number of areas where current OS design should be amended. Existing micro-kernels support resources such as protection domains, threads, regions of VM backed by external mappers, and communication endpoints. Implementation of object-support subsystems does use these resources; however:

- Their interface should be unified, to facilitate the work of the subsystems.
- Micro-kernels typically allocate a name (a UID) to each resource. This is a waste and an encumbrance to subsystems, which have their own naming.
- Object-support subsystems need support for grouping and demultiplexing ressources.

We have exposed our views on the above issues in previous papers [7, 8]. We here turn to more specific adaptations of the micro-kernel technology, which are the current focus of our work: A layer of common distributed abstractions, above the kernel, and a lightweight, uniform, garbage-collected reference mechanism.

2 Library of Common Abstractions

Currently, subsystems (such as a Unix subsystem or an object-support subsystem [2]) are built directly on top of the microkernel. Abstractions common to multiple subsystems, but too high-level to belong in the microkernel interface, cannot be shared. Therefore, we are currently designing and implementing a library of useful distributed abstractions, BOAR [5]. BOAR is structured as a layered set of Fragmented Objects [3, 4], a powerful distributed structuring model. Subsystem implementors will find in BOAR a set of ready-made tools. OS designers will like it because of its clean, orthogonal layers:

- The base layer is a set of lightweight protocols for passing data and control independently (see next section), with two implementations, using shared plysical memory and messages.
- Different transport protocols are implemented above the base layer: datagram, RPC, unreliable group multicast, causality-preserving messaging. Implementations are optimized both for shared memory and for networks.
- Using the transport protocols, a collection of useful distributed data management abstractions: coherency protocols, concurrency control protocols, and persistency protocols.
- Using the coherency protocols, multiple flavors of distributed shared memory. Again, implementations are optimized both for networks and for shared physical memory.

Architecturally, the main effect of the library approach is to blur the frontier between micro- and macro-kernel.

3 Lightweight, Uniform, Garbage-Collected, Distributed References

Existing microkernels support *ports* or communication endpoints. Although lighter than their Unix counterparts (pipes and sockets) they are still too coarse. For instance, a Chorus port is all these mechanism at once, wired into a single interface: naming, binding, location, protection, a communication protocol, queuing of data, and scheduling of control. Even worse, a Mach

port adds in a kernel-interpreted presentation protocol, and a termination protocol.

We believe that breaking this into orthogonal components has the potential for cleaner, clearer semantics, and for better performance. Bershad has already demonstrated this for the data and scheduling components [1]. We are currently working on the naming, binding and location components. We propose OS-supported, lightweight, fine-grain *references.*

A reference is the OS equivalent of pointers. A reference may be distributed (designating a potentially remote object) and/or persistent (between stored objects, or between a stored and an in-memory object).

A reference is formed by a sequence of *links* bound into a *chain.* Each individual link implements a very primitive piece of functionality. It is composed of a few bytes of opaque data, along with a very short sequence of code (down to a few assembly instructions) that interprets the data. This architecture allows to uniformly represent very different link types: within an address space, cross-space, or cross-machine.

Manual management of distributed and/or persistent shared data can be horribly complex. Therefore our references support automatic, distributed garbage collection [6, 10]. The GC protocol is a conservative variant of reference counting, augmented to deal with cyclic garbage.

A reference can be stored on disk or passed in a message. We specify a straightforward presentation protocol which allows the garbage collector to keep track of all references. The presentation protocol (i.e. marshalling/unmarshalling code) executes within the application context, with no message or system call overhead, and no kernel involvement.

Occasionally, longer-than-necessary chains will form. These are short-circuited in a lazy fashion by a separate binding and location protocol.

4 Conclusion

The trend towards microkernels has been beneficial in identifying small, orthogonal, common abstractions needed by applications, and exposing their power to knowledgeable designers. This has been a long, painstaking process. Our work refines on this, building a library of useful medium-level abstractions, offering a rich, layered environment from which application designers can pick and choose.

We have also identified a weakness of current microkernel designs: ports are too heavyweight, bundling too many mechanisms together. Instead we propose to break them into their individual components: passing data, passing control, naming, binding, location. Our references are chains of very simple links, and support garbage collection.

References

[1] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[2] Sabine Habert, Laurence Mosseri, and Vadim Abrossimov. COOL: Kernel support for object-oriented environments. In *ECOOP/OOPSLA'90 Conference*, volume 25 of *SIGPLAN Notices*, pages 269–277, Ottawa (Canada), October 1990. ACM.

[3] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Structuring distributed applications as fragmented objects. Rapport de recherche 1404, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), January 1991.

[4] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Fragmented objects for distributed abstractions. In T. L. Casavant and M. Singhal, editors, *Advances in Distributed Computing: Concepts and Design*. IEEE Computer Society Press, 1992. To appear.

[5] Mesaac Makpangou, Yvon Gourhant, and Marc Shapiro. BOAR: A library of fragmented object types for distributed abstractions. In *Proc. of the International Workshop on Object-Orientation in Operating Systems*, Palo Alto, CA (USA), October 1991.

[6] Marc Shapiro. A fault-tolerant, scalable, low-overhead distributed garbage detection protocol. In *Tenth Symp. on Reliable Distributed Sytems*, Pisa (Italy), October 1991.

[7] Marc Shapiro. Object-support operating systems. *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems and Application Environments*, 5(1):39–42, 1991.

[8] Marc Shapiro. Soul: An object-oriented OS framework for object support. In *Workshop on Operating Systems for the Nineties and Beyond*, Dagstuhl Castle, Germany, July 1991. Springer-Verlag.

[9] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating system — assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.

[10] Marc Shapiro, Olivier Gruber, and David Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Rapport de Recherche 1320, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), November 1990.

Reliable Multicast between Microkernels

Robbert van Renesse <rvr@cs.cornell.edu>
Ken Birman <ken@cs.cornell.edu>
Robert Cooper <rcbc@cs.cornell.edu>
Bradford Glade <glade@cs.cornell.edu>
Patrick Stephenson <patrick@cs.cornell.edu>

Computer Science Department, Upson Hall
Cornell University, Ithaca, NY 14853

ABSTRACT

ISIS is a system for building applications consisting of cooperating, distributed processes. Here we present a new implementation of the ISIS system, geared towards modern microkernel technology. We have adopted similar strategies, such as using basic internal mechanisms for efficiency, external services to implement policies, and light-weight user space constructs for simplicity. The resulting design is less complex and more efficient than the present one. This discussion focuses on the integration of our new system into the MACH and Chorus systems, and discusses the status and performance of an initial implementation.

1. Introduction

ISIS [1, 2], developed at Cornell University, is a system for building applications consisting of cooperating, distributed processes. Group management and group communication are two basic building blocks provided by ISIS. ISIS has been very successful, and there is currently a demand for a version that will run on many different environments and transport protocols, and will scale to many process groups. Furthermore, performance is an important issue. For this purpose, ISIS is being redesigned and rebuilt from scratch [3].

Of particular importance to us is getting the new ISIS system to run well on modern microkernel technology, notably MACH [4] and Chorus [5, 6]. These systems

The authors were supported under DARPA/NASA grant NAG-2-593, a grant of the Dutch Royal Academy of Sciences (KNAW), and by grants from IBM, HP, Siemens, GTE, and Hitachi.

provide their own communication mechanisms on top of an interface based on ports and messages. We wish to integrate the ISIS system within this framework. The basic reasoning behind these plans is that microkernels appear to offer satisfactory support for memory management and communication between processes on the same machine, but that support for applications that run on multiple machines is weak. The current IPC mechanisms, with Remote Procedure Call as the most popular one, are adequate only for the simpler distributed applications [7, 8, 9].

The new ISIS system has several well-defined layers. This paper discusses each layer in turn. The lowest layers, which implement multicast transport and failure detection, are near completion and currently run on SUN OS using SUN LWP threads, on MACH using C Threads, and on the x-kernel [10]. This system can use several different network protocols at the same time, such as IP, UDP, Deering multicast [11], and raw Ethernet. This enables processes on SUN OS, on MACH, and on the x-kernel to multicast among each other, even though the environments are very dissimilar.

We are currently in the process of extending the native interfaces with group semantics. Chorus already provided a group notion, which we are able to support with few interface changes but much stronger semantics. In MACH, we have introduced a new notion, the group port. Unlike the normal MACH port, several processes may appear to have receive rights on the same group port. There are defaults so that existing applications may run unmodified and not know that they are broadcasting rather than sending messages point-to-point. A group port maps to an ISIS group address, and messages sent by MACH to a group port may be received by ISIS processes running on different operating systems, and the other way around. This introduces some protection problems, which are being addressed in other work [12].

The system makes use of available hardware multicast if possible. It also queues messages if a backlog appears, so that multiple messages may be packed together in a single packet. Using this strategy, the number of messages per second can become very large, and in the current (simple) implementation about 10,000 per second can be sent between distributed SUN OS user processes, a figure that approaches the speed of local light-weight remote procedure call mechanisms.

This paper starts by presenting *ports* as a communication interface in Section 2. Section 3 describes the new ISIS system, and Section 4 how this system may be customized to use the port interface. Section 5 describes the new, layered design, and how this corresponds to microkernel technology. In Section 6 we focus our attention on supporting multi-threading. Section 7 presents a portable and light-weight construct for group management and communication. Section 8 discusses issues raised by the need to support the old ISIS toolkit over the new system. We present the current status of the implementation, along with some initial performance results in Section 9. In Section 10 we offer some concluding remarks.

2. Ports

Modern operating systems support ports with a wide range of semantics. As a minimum, as in the Amoeba system [13], a port is an address to which messages can be sent. In Chorus and MACH, a bounded queue is associated with the port, so the process holding the port need not listen continuously (in Amoeba this is done by having several threads wait for messages simultaneously). In MACH, the messages are reliably delivered, and the sender may block if the port queue is full. Chorus messages to ports are unreliable, although a reliable port-level RPC interface is supported. MACH and Chorus allow only one receiver process on a port (although possibly multiple threads within that process), but the port may be migrated to a new receiver process.

MACH ports do not have user-visible global names, and have, in reality, no global access. The ports, instead, are accessed using so-called rights, which can be compared to file descriptors or capabilities. Global access is simulated through a user space server, the NetMsgServer. This server acts as an agent for remote ports: it creates a local proxy port, and forwards messages sent to the port to the remote NetMsgServer using TCP or another conventional protocol. Similarly, it delivers messages received from remote NetMsgServers to the local port. MACH users do not notice this, and, in principle, local semantics are transparently maintained (currently, however, this is not the case). Chorus ports, on the other hand, do have global names and a corresponding global implementation.

Chorus provides a port group concept, with weak semantics. It is possible to allocate a group, and add (local) ports to it. Messages can be sent to the group in the same way as to ports, and are unreliable. There is no membership information available. (Note: the MACH ''port set'' is a a mechanism that allows receiving on multiple ports at once, much like UNIX† *select*, and has no group communication role. MACH currently does not provide a group mechanism.)

3. The new ISIS system

The ISIS system, as it is currently distributed and supported, is the result of a long evolution. As such, it supports many different applications, and has grown rather complex. The time has come to re-implement the system using the experience gained. It appears that a design strategy such as that used in microkernel-based operating systems is applicable here: a lean core system provides the basic functionality, on which more complicated services can be built. For example, much as a pager can be implemented as an external server using basic mechanisms in the kernel, so can a failure detector run in user space as a service for a communication system. The advantage of this approach is that it

† UNIX is a Registered Trademark of AT&T in the USA and other countries.

makes it possible to experiment with new failure detection mechanisms without major disruption of the system. Moreover, in contrast to an external pager (which might see extremely heavy, performance-critical traffic), the modules external to our new system are infrequently used ones that are off the critical path for the most common communication patterns.

The basic functionality that we plan to put in the core of the system is most easily illustrated through comparison with the USENET news service. The new system will support ''ISIS news groups,'' to which ports can subscribe. Unlike USENET, ISIS news groups may be created dynamically, and, as will be seen, they have stronger semantics. Processes may post messages to news groups, which will be reliably delivered to all subscribing ports. The facility presents the same interface in local-area and wide-area settings. It is possible to reply to messages or to follow-up on messages, just like in USENET. Unlike USENET, it is impossible to receive a follow-up before the original message (an annoying feature of USENET). This guarantee is known as *causality* (see Figure 1). This causality guarantee works even across multiple news groups.

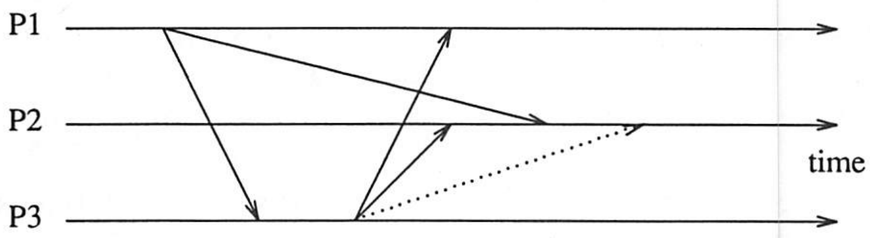


Fig. 1. Causal ordering. In this case we have a news group with 3 subscribers. Subscriber P1 posts a message which is sent to P2 and P3. P3 picks up the message and follows up on it. Due to unexpected network delays, it is possible that P2 receives the follow-up message before the original message from P1. ISIS guarantees that this reversal does not happen, so that the message from P3 to P2 follows the dotted path.

An important distinction from the USENET model is that ISIS news supports a way to track membership in ''news groups.'' If so desired, membership changes will be posted to the news group just like other messages. These changes include ports subscribing, unsubscribing, failing, or becoming unreachable. Notification of these changes are causally ordered with respect to normal messages. Therefore, after the notification that a port has unsubscribed or failed, it is impossible to receive messages from it. All subscribers see exactly the same history of membership changes. In addition, it is possible, using cryptographic protection, to preclude processes from subscribing or posting to groups [12]. This kind of functionality simplifies the development of secure, fault-tolerant tools and applications.

In current ISIS applications, groups are used heavily. For example, replicated ob-

ject repositories generally use a group per object. The subscribers of each group are the servers that store a replica of the object, or have a cached copy, since each server is interested in updates to the object. The ordering semantics on messages and group membership make it easy to maintain consistency among the copies and to divide work over the servers to enhance performance.

Using the new ISIS system, existing technology like RPC and atomic transactions may be supported easily, and with much simpler and more consistent error semantics than is usually the case in distributed computing environments. Thus ISIS should be seen as an orthogonal paradigm which provides a simple message interface with simple, but strong error semantics. Interestingly, we think that we can achieve comparable performance in message delivery, since our protocols (if failures are infrequent) rarely introduce extra messages. Moreover, since the ISIS interface is asynchronous in nature, we hope to provide a much higher message throughput (messages/time unit) than other communication mechanisms, as we will be able to pack multiple messages in a single packet. Our initial results show that we are able to achieve at least 10,000 messages per second on normal Ethernet technology.

4. Customizing ISIS to a port interface

The original ISIS runs as a UNIX application, and so is available to both MACH and Chorus users. In the new ISIS system, we want to implement the core ISIS mechanisms as extensions to a set of similar, but different host microkernel operating systems, in particular MACH and Chorus. That is, we want to add stronger functionality and semantics to the existing MACH and Chorus message interfaces, rather than defining a new interface. This functionality and semantics will take the form of ISIS news groups.

The ISIS news interface is sketched in Figure 2. *Port.send* and *port.receive* are the unchanged interface to the existing port interface. *Ng.create* creates a new news group. The options specify the kind of ordering required (unordered, FIFO, causal, or total), whether or not membership information need be posted, the degree of fault-tolerance (*e.g.*, whether logging is desired or not), which network transport mechanism to use, and possibly more. *Ng.subscribe* and *ng.unsubscribe* allow ports to be added to and removed from news groups.

The news interface will look the same on both Chorus and MACH. However, we also want to extend the basic Chorus and MACH system calls so that we can transparently replace a conventional Chorus or MACH application with one that has been replicated for fault-tolerance. The users of such an application would continue to use the original application interface, but all communication would transparently be sent to a group, and processed cooperatively by the group members. This raises the question of how to integrate the ISIS news group mechanisms into the Chorus and MACH microkernel interfaces.

procedure	*arguments*	*result*
ng.create	options	news group
ng.subscribe	port, news group	
ng.unsubscribe	port, news group	
ng.destroy	news group	
port.send	port, message	
port.receive	port	message

Fig. 2. The basic port interface to ISIS.

As described, Chorus currently provides an unreliable group interface. The Chorus group-allocate function will be implemented by creating an ISIS news group. A new option will indicate if membership information should be posted to the group or not. Adding a port to the group will be the same as subscribing the port to the news group [14].

MACH does not currently have a group concept. (As noted before, the ''port set'' provides only a select functionality, with no global semantics.) We intend to use the NetMsgServer to simulate a global port with multiple receivers, much in the same way as MACH already uses the NetMsgServer for simulating global ports. Instead of TCP, the NetMsgServers will use ISIS to do their group communication (see Figure 3). Each application task will have two ports: one to receive incoming messages, and another to which it sends outgoing messages. The NetMsgServer forwards messages it receives from ISIS to the first port. Messages sent to the second port are received by the NetMsgServer and forwarded to the corresponding news group [15].

It may be useful to walk through this architecture in the case where a pre-existing application is replaced with a fault-tolerant group. This will work roughly the same way under MACH and Chorus. Under MACH, when a client thread looks up the name of a server to find the port, this request is sent to the NetMsgServer. The NetMsgServer can now generate a local port and return the send right to the client. The NetMsgServer subscribes the port to the ISIS news group that implements the fault-tolerant service. When the client sends a request message to the port, the NetMsgServer will post this to the news group, the members of which cooperate to respond fault-tolerantly using one of several methods supported by Isis. After the NetMsgServer receives a reply to the request, it forwards it back to the client. The client can be kept completely unaware of this change. (Details of the comparable algorithm in Chorus are omitted for brevity.)

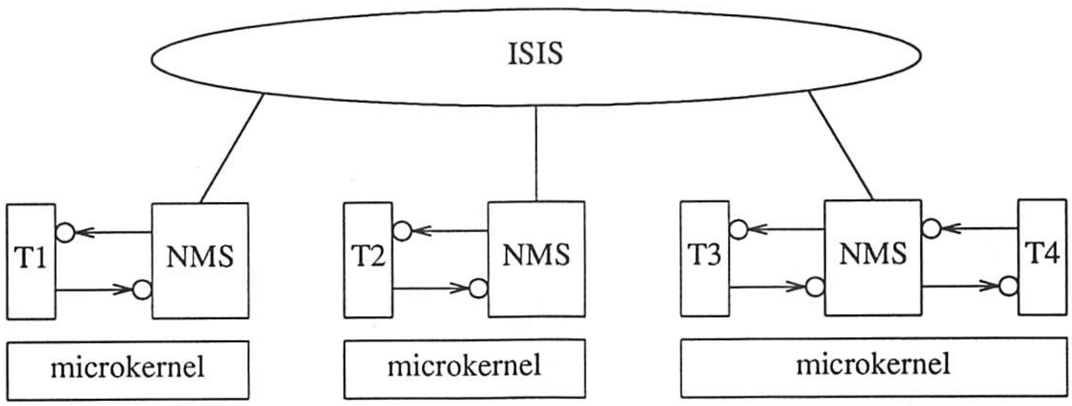


Fig. 3. Structure of the MACH/ISIS implementation with four tasks, showing the microkernels, the NetMsgServers (NMS), and the ports.

5. Design

The new ISIS implementation, now under development, follows the design principles of microkernel operating systems closely. It consists of a small core that implements the basic protocols. More policy-oriented matters, such as the determination of what constitutes a failure and how to detect one, or how to log messages in fault-tolerant settings, are implemented externally. This compares to similar concepts in microkernel designs, such as external pagers and schedulers. In this section we will describe the new design inside-out (see Figure 4). Usually, this would correspond to bottom-up, as the core would be implemented near the microkernel, and the external services in user space.

At the very core of the new system are the network protocols that are available for the host system. Typically, these would include IP and UDP, optionally extended to include the Deering multicast facilities. In addition, there may be access to the raw network hardware, or to more sophisticated protocols that may implement forms of reliable multicast.

At the first ISIS shell, we provided a layer called the Multicast Transport Service (MUTS) [3, 16]. MUTS provides a simple, reliable multicast service that runs on one or more lower level protocols. It does not implement any group membership services, and only provides FIFO ordering. Thus ''follow-ups'' from a third party may still arrive at a particular destination before the original message. MUTS detects communication problems, but does not act upon them other than reducing the retransmission rate to a problematic member. Instead, it passes error statistics up, in the hope that external layers will decide to delete the member from the group. Finally, it supports optional encryption and signing of messages to support security.

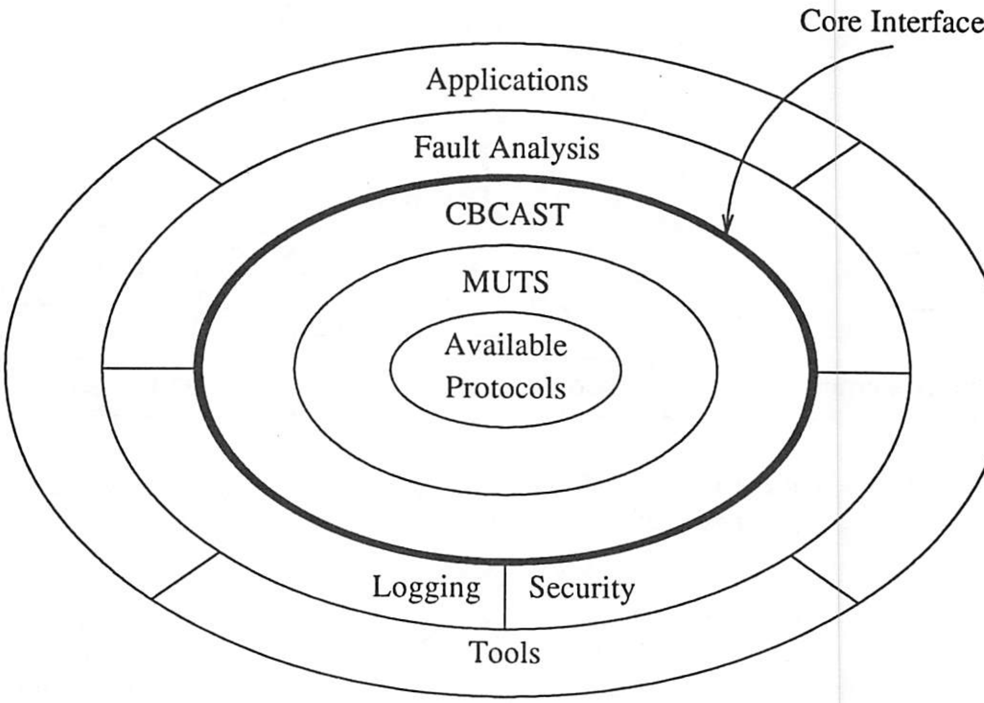


Fig. 4. Structure of the new ISIS implementation

The second shell is the Causal Broadcast (CBCAST) layer. It implements group membership protocols and guarantees causal ordering of messages. Causal ordering is achieved, in principle, by attaching a simple vector of sequence numbers to each message that allows receivers to detect if they have missed any messages [17]. If so, they will delay the delivery of the message. In practice it is usually unnecessary to attach the whole vector, since changes are mostly minimal. CBCAST does not actually decide on failures, but relies on an incoming stream of failure notifications from an external source. CBCAST merges this stream with the message stream, and uses it to provide a consistent view of the group membership. The CBCAST interface is the core interface, and both MUTS and CBCAST will usually be located near the network interface for efficiency.

The next shell contains the external services, which implement policies rather than mechanisms. These services may use the core mechanisms, but need to be careful not to create infinitely recursive dependencies. Other than the core services, they can make use of local operating system facilities such as disk storage. The first external service, the Failure Detector (FD), performs fault analysis [18]. FD is a distributed service that uses information passed up by MUTS, in addition to information made available by the host operating system, to determine the sites that have become unreachable. As its output, it provides a stream of failure notifications. Each member site of FD will provide the same

stream, and feed it to its local CBCAST shell. As it is an external service, users may plug in their own failure detectors for special applications.

A second external service is the logging facility. As discussed earlier, this facility logs messages that have been posted to a news topic on stable storage. The facility is only needed in applications that desire long-term stability of messages.

The last external service assists in providing security. MUTS can encrypt and sign messages, but is not involved in key distribution itself. Instead, this is done with the help of an authentication service [12]. When security has to be provided right from a cold start, this service also assists in bootstrapping the system.

The outermost shell consists of the applications, and standard tools for implementing distributed applications. ISIS provides a set of those tools, such as RPC, atomic transactions, monitoring and control facilities, and resource management, which will be described in Section 8. Additionally, this layer provides a light-weight group mechanism which simplifies resource allocation and provides a portable interface to heterogeneous systems. This light-weight group mechanism is the subject of Section 7.

6. Multi-threading and ordering

The current ISIS system is multi-threaded. For each arriving message, a new thread is created. ISIS was one of the first available systems under the UNIX system that provided multi-threading, and this was one of the reasons for its success. Now that microkernels provide their own multi-threading, we wish to adopt the available threading mechanisms, rather than impose our own. Yet there is a problem to be resolved.

To maximize concurrency, an application may want to have several threads in a process reading from the same port. When a message becomes available, a randomly selected thread will get the message. If multiple messages become available at once, several threads will get a message, one each, and start processing the messages in parallel. As a consequence, ordering is lost. In the current ISIS this is solved by scheduling threads non-preemptively and in the correct order. However, if we want to be able to take full advantage of real parallelism, this approach no longer suffices.

Consequently, the new ISIS system attaches a sequence number to each message. The MACH interface already supports this, and the Chorus interface can easily be extended to support a sequence number in its message descriptor. Any ordering issues are now left to the application. To support this, the ISIS library provides a construct called *event counters*, based on Reed and Kanodia's work on synchronization [19]. An event counter is basically a lock, which can be acquired only in a certain order. The interface is presented in Figure 5.

Ec.create creates a new event counter, and initializes it to zero. To acquire the event counter, a thread calls *ec.acquire* with a sequence number. If the sequence number

procedure	*arguments*	*result*
ec.create		event counter
ec.acquire	event counter, sequence number	
ec.release	event counter	
ec.destroy	event counter	

Fig. 5. The event counter interface.

does not correspond with the event counter, the procedure will block until the value of the event counter has reached the given value. *Ec.release* will release the event counter, and increment its associated value. *Ec.destroy* will destroy the event counter, and release the associated resources. Figure 6 demonstrates how an event counter may be used. Basically, an event counter implements a critical region for processing messages, so that the messages are processed in the right order.

```
thread ( port, event counter ) {
        for ever {
                port.receive ( port ) → { message, sequence number };
                initial processing on message;
                ec.acquire ( event counter, sequence number );
                main processing on message;
                ec.release ( event counter );
                final processing on message;
                release message;
        }
}
```

Fig. 6. How event counters are used. Several of these threads may run in parallel, yet the main processing on messages happens in a strict order.

This simple interface is consistent with the microkernel philosophy underlying our work. Other researchers have proposed more elaborate interfaces for this type of event ordering. For example, the PSYNC system allows programs to detect and act upon very complex message ordering properties [20], and the work of Liskov and Ladin also supports user-implemented message delivery orderings [21]. Experience with ISIS, however, leads us to believe that while causal delivery is vital, other sorts of delivery orderings are rarely needed. The approach described above, which forces users to process mes-

sages in a fixed order consistent with causality is less powerful than these other schemes, but it has the benefit of being simple and highly concurrent. Single-threaded application may ignore event counters altogether.

7. Light-weight groups

As mentioned earlier, ISIS users have found it convenient to implement small, single objects as separate groups, rather than having one group that manages a complete service with a large set of objects. For example, in a replicated file system, a file would be a group with a member on every server that stores the file. For each open file, there would be a group that includes not only the servers, but also the clients that maintain cached copies of the file. To deal with the resulting proliferation of groups, we saw a need for a light-weight version of the group mechanisms.

Again, we were influenced by microkernel design concepts, in which several light-weight mechanisms are provided in user space. The most obvious of these is the light-weight process or thread abstraction. Another well-known, older abstraction is memory allocation. These abstractions not only allow easier resource management by sharing most of a core environment, but also provide a portable interface across different environments. For example, POSIX threads and *malloc* provide a portable, light-weight execution and memory allocation facility which may be used for applications that need to run on several different platforms.

In the new ISIS system we have designed a portable, light-weight news group (LWG) mechanism. The basic idea is that many LWGs will be mapped to a single news group as implemented by the core ISIS system. Thus, these LWGs will share the same security environment, and have the same failure model, as their messages will be multiplexed over a single core news group. The benefit of this approach is that membership changes to the news group will now automatically affect a potentially large number of LWGs, allowing us to amortize the cost of maintaining membership information over what the application thinks of as being a large number of independent groups. The old ISIS system lacked such a facility, forcing many application programmers to develop equivalent mechanisms on their own. The interface to light-weight groups is the same in MACH, Chorus, or any other system that supports ISIS, and is listed in Figure 7. Instead of an *lwg.receive* interface, the user provides an upcall to *lwg.subscribe* (an upcall interface fits the asynchronous model of ISIS better). The upcall is invoked each time a message for the LWG arrives, with the message and an event counter as arguments.

Applications using this interface, along with a standard thread interface (also provided in the ISIS library) and the UNIX interface, should run equally well under MACH or Chorus. As there is nothing that prevents MACH and Chorus processes from subscribing to the same ISIS group, applications should even be able to run on a mixture of these systems. There is a complication though, since the network message formats for MACH

procedure	*arguments*	*result*
lwg.create	core news group	lwg
lwg.subscribe	lwg, upcall	
lwg.unsubscribe	lwg	
lwg.send	lwg, message	
lwg.destroy	lwg	

Fig. 7. The light-weight group interface.

and Chorus are different (MACH messages are self-describing). To make this work, we intend to provide an option to the create function of core news groups that results in ''raw'' mode: in this mode, messages pass through the port interface unchanged as byte arrays. (In MACH, it would clearly be impossible to distribute port rights through this type of group, although rights could still be sent through separate ''pure Mach'' groups.)

8. The ISIS toolkit

Readers familiar with the ISIS system will have noticed that several aspects of the system have not been described yet, in particular the tools. The current ISIS system comes with a *toolkit* containing tools for applications that use RPC, atomic transaction, primary-backup replication, resource management, and monitoring and control [22]. The new system will continue to support these tools, but they will be implemented entirely as a set of user libraries and services. Other old ISIS applications will continue to be supported through a compatibility library that offers the old interface. For brevity, we will not present details of this mapping. The key point, however, is that the CBCAST and group membership layer of our new system is sufficiently powerful to let us support the full range of functionality implemented by the old ISIS system.

9. Status of the implementation

The new ISIS system is in a fairly advanced stage of implementation. The MUTS shell has been completed, and currently runs on several different platforms, namely SUN OS in user space (using SUN LWP threads), MACH 2.5 and 3.0 in user space, and the x-kernel in kernel space [10]. This implementation uses IP or UDP, optionally extended with the Deering multicast facilities [11], or raw Ethernet. A simple version of the CBCAST shell has been implemented, but does not fully support multiple groups and failure handling. We also have an initial version of the failure detector running.

MUTS uses a sliding window protocol adapted for multicast delivery. In this protocol, the send window is not moved up until acknowledgements have been received

from all destinations. However, acknowledgements do not have to be sent for every packet, but only when the send window fills up. Also, acknowledgements may be piggybacked on other traffic. To cope with many destinations deciding to acknowledge a window of packets simultaneously, it is possible to have different destinations acknowledge at different times.

Because the ISIS interface is asynchronous, the bandwidth of messages may be higher than the underlying protocols can handle. If this occurs, MUTS will start queuing up messages. When a slot becomes available in the send window, MUTS fills up the packet with as many messages as possible. This way, the number of messages can become much larger than in conventional, synchronous systems. In our initial implementation, we achieve 10,000 (1-byte) messages per second from SUN OS user process to user processes on different SPARCstation 2s connected by Ethernet, and about 5,000 parallel RPCs per second using a simple RPC implementation on top of the MUTS interface. The maximum data bandwidth is about 800,000 bytes per second, using larger messages (which may be up to 2 Gigabytes). Initial results on the light-weight group mechanisms (on the old ISIS system we achieved a speed-up of about 9 for the group create operation and even higher speedup factors for other operations) lead us to believe that we will be able to produce a fast system.

In the MACH and Chorus systems, the plan is to run both MUTS and CBCAST as x-kernel protocols. In MACH, the NetMsgServer will be reimplemented to use the x-kernel, so that the core of the ISIS system will run within the NetMsgServers. In Chorus, the x-kernel will be made part of the microkernel itself. We expect to provide the first releases of these implementations by the end of 1992.

10. Conclusion

We have described a new architecture for the ISIS system, geared towards use in microkernel-based operating systems. We have used the same design concepts as were used for these operating systems. These concepts include a clean division of work between the kernel and user space responsibilities and light-weight constructs. The resulting system is elegant, efficient, and easily customized to the environment in which it has to operate. An initial implementation of the new system is nearly finished, and shows promising performance, such as 10,000 messages per second between user processes on different sites. We hope to have a usable system running on MACH, Chorus, and the x-kernel by the end of 1992.

Acknowledgements

We would like to thank Aleta Ricciardi, Carlos Almeida, Micah Beck, and Mike Reiter for their helpful comments, and the MACH, Chorus, and x-kernel people for interesting discussions and elucidations.

References

[1] K. Birman and R. Cooper, ''The ISIS Project: Real Experience with a Fault Tolerant Programming System,'' *Operating Systems Review*, pp. 103-107, April 1991.

[2] K. P. Birman and T. A. Joseph, ''Exploiting Virtual Synchrony in Distributed Systems,'' *Proc. of the 11th ACM Symp. on Operating Systems Principles*, pp. 123-138, Austin, TX, November 1987.

[3] The ISIS Group, ''The Restructuring of ISIS for Modern Distributed Operating Systems,'' Internal Cornell Report, September 1991.

[4] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, ''Mach: A New Kernel Foundation for UNIX Development,'' *USENIX Summer '86 Conf. Proc.*, pp. 93-112, Atlanta, GA, June 9-13, 1986.

[5] F. Armand, M. Gien, F. Hermann, and M. Rozier, ''Revolution 89, or Distributing UNIX Brings it Back to its Original Virtues,'' *Proc. of the Workshop on Experiences with Building Distributed (and Multiprocessor) Systems*, pp. 153-174, Ft. Lauderdale, FL, October 5-6, 1989.

[6] M. Rozier et al, ''Chorus Distributed Operating System,'' *Computing Systems*, Vol. 4, No. 1, 1988.

[7] K. G. Hamilton, ''A Remote Procedure Call System,'' Ph.D. dissertation, Tech. Rep. no. 70, Computing Laboratory, University of Cambridge, Cambridge, England, December 1984.

[8] H. E. Bal, R. van Renesse, and A. S. Tanenbaum, ''Implementing Distributed Algorithms using Remote Procedure Call,'' *Proc. of the 1987 National Computer Conf.*, pp. 499-506, Chicago, IL, June 15-18, 1987.

[9] A. S. Tanenbaum and R. van Renesse, ''A Critique of the Remote Procedure Call Paradigm,'' *Proc. of the EUTECO 88 Conf*, pp. 775-783, ed. R. Speth, Vienna, Austria, April 20-22, 1988.

[10] N. Hutchinson and L. Peterson, ''The x-Kernel: An Architecture for Implementing Network Protocols,'' *IEEE Transactions on Software Engineering*, Vol. 17, No. 1, January 1991.

[11] S. E. Deering and D. R. Cheriton, ''Multicast Routing in Datagram Internetworks and Extended LANs,'' *ACM Trans. Comp. Syst.*, Vol. 8, No. 2, May 1990.

[12] M. Reiter, K. Birman, and L. Gong, ''Integrating Security in a Group Oriented

Distributed System,'' *IEEE Symposium on Research in Security and Privacy*, Oakland, CA, May 4-6, 1992.

[13] A. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. Mullender, A. Jansen, and G. van Rossum, ''Experiences with the Amoeba Distributed Operating System,'' *Comm. ACM*, Vol. 33, No. 12, pp. 46-63, December 1990.

[14] M. Beck, K. Birman, R. Cooper, and S. Toueg, ''A Fault Tolerant Extension of the Chorus Nucleus,'' Internal Cornell Report, January 1991.

[15] R. van Renesse, K. Birman, B. Glade, and P. Stephenson, ''Options for Adding Group Semantics to Ports,'' Internal Cornell Report, January 1992.

[16] R. van Renesse, ''A MUTS Tutorial,'' Internal Cornell Report, September 1991.

[17] K. Birman, A. Schiper, and P. Stephenson, ''Lightweight Causal and Atomic Group Multicast,'' *ACM Transactions on Computer Systems*, Vol. 9, No. 3, pp. 272-314, August 1991.

[18] A. Ricciardi and K. Birman, ''Using Process Groups to Implement Failure Detection in Asynchronous Environments,'' *ACM Symposium on Principles of Distributed Computing*, pp. 341-353, Montreal, Quebec, Canada, August 19-21, 1991.

[19] D. P. Reed and R. K. Kanodia, ''Synchronization with Eventcounts and Sequencers,'' *Comm. of the ACM*, Vol. 22, No. 2, pp. 115-123, February 1979.

[20] L. Peterson, N. Bucholz, and R. Schlichting, ''Preserving and Using Context Information in Interprocess Communication,'' *ACM Transactions on Computer Systems*, Vol. 7, No. 3, pp. 217-246, August 1989.

[21] B. Liskov and R. Ladkin, ''Highly-Available Distributed Services and Fault-Tolerant Distributed Garbage Collection,'' *Proc. of the Fifth ACM Symp. on Principles of Distributed Computing*, pp. 29-39, Calgary, Alberta, August 1986.

[22] K. Marzullo, R. Cooper, M. Wood, and K. Birman, ''Tools for Distributed Application Management,'' *IEEE Computer*, Vol. 24, No. 8, pp. 42-51, August 1991.

Designing a Scalable Operating System for Shared Memory Multiprocessors

Michael Stumm, Ron Unrau, and Orran Krieger

Department of Electrical Engineering
University of Toronto
Toronto, Canada M5S 1A4
Email: stumm@csri.toronto.edu

Abstract

We propose a simple structuring technique based on *clustering* for designing scalable shared memory multiprocessor operating systems. Clustering has a number of advantages. First, it encorporates structuring principles from both tightly-coupled and distributed systems, and attempts to exploit the advantages of both. Second, it maximizes locality, which is key to good performance in large systems. Finally, clustering simplifies the design of large-scale operating systems and makes the scalability aspect of the system portable to different architectures.

In this paper, we describe clustering, how it is applied to a micro-kernel-based operating system in general, and how it affects the design and implementation of the memory management subsystem in particular. We present preliminary measurements of performance, obtained from a working prototype implementation.

1 Introduction

Considerable attention has been directed towards designing "scalable" multiprocessor hardware, capable of accommodating a large number of processors. These efforts have been successful to the extent that a number of such systems exist, some in prototype form. However, scalable multiprocessor hardware is cost effective only if the operating system is as scalable as the hardware (and the same is true for parallel applications).

This paper addresses the issue of scalability in operating system design for shared memory multiprocessors, such as the Stanford Dash [17], the Kendall Square Research KSR-1 [8], the BBN TC2000 [6], the IBM RP3 [19], and University of Toronto's Hector [20]. Typically, existing multiprocessor operating systems have been scaled to accommodate a large number of processors in an ad hoc manner, by repeatedly identifying and then removing the most contended bottlenecks. This is done either by splitting existing locks, or by replacing existing data structures with more elaborate, but concurrent ones. This can be a long and tedious process, resulting in systems that 1) are fine-tuned for a specific architecture; and hence are not easily portable to other hardware bases with respect to scalability; 2) are not scalable in a generic sense, but only until the next bottleneck is reached); and 3) have a large number of locks that need to be held for common operations, with correspondingly large overhead.

We believe that a more structured approach is necessary to achieve scalability that is portable, generic, and tunable. In this paper, we propose and describe a simple structuring technique to achieve scalability based on *clustering*. Clustering encorporates structuring principles from both tightly-coupled and distributed systems, and attempts to exploit the advantages of both. On the one hand, the structure corresponds to that of a distributed system with no central servers — these systems have proven to scale reasonably well in some instances.On the other hand, there is tight coupling within a cluster, so the system is expected to perform well for the common case, where interactions occur primarily between objects located in the same cluster. The structure assumes a base consisting of a micro-kernel together with server processes, which can easily be replicated for increased parallelism, and which provides for locality in a natural way. We describe clustering in Section 3.

We illustrate and test our ideas on is an experimental operating system called *Hurricane* we have written for the Hector Multiprocessor, but we argue that clustering could be applied equally well to any micro-kernel-based system. Hurricane has been running on the Hector multiprocessor for close to two years now; the Hurricane kernel is similar to the V kernel, and in fact supports essentially the same source-level interface. We give a brief overview of Hurricane in Section 4, where we also describe how clustering is applied and the advantages it brings. In Sections 5 and 6 we describe in detail how clustering is applied to individual specific components of the operating system, namely Hurricane's facilities for process management and interprocess communication (i.e. message passing), and the single-level-store memory management system. Finally, in Section 7 we present performance measurements obtained from our implementation. First, however, it is illustrative to consider the meaning of scalability and its implications.

2 Requirements for Scalability

It is surprisingly difficult to give a formal definition that characterizes scalability in a general sense, especially for operating systems. For example, one of the better known formal definitions of scalability is by Nussbaum and Agarwal[1], but is not applicable to operating systems for several reasons. First, their definition considers the operating system to be either an extension of the hardware, or an extension of the (application) algorithm. Second, the definition is based on asymptotic limits and thus provides no hints as to how to design or build a scalable (operating) system. Finally, the definition is based on speedup, and hence the response time, which we do not believe to be appropriate for the operating system domain. For parallel systems, we do not intuitively expect the response time of an operating system call to speed up as more processors are added to the system; rather, we are content if the system call does not take longer to complete as additional processors are added. Instead, we believe that throughput and concurrency are the dominant issues in designing scalable operating systems. Since we expect the demand on the operating system to increase proportionally to the size of the system, the throughput must also increase proportionally, which is possible only if the operating system is highly concurrent.

Instead of attempting to construct a new definition, we identify three requirements we believe are necessary for any operating system to be scalable:

[1] Nussbaum and Agarwal's definition is: *The scalability of a machine for a given algorithm is the ratio of the asymptotic speedup on the real machine and the ideal realization of an EREW PRAM* [18].

Preserving parallelism: *The operating system must preserve the parallelism afforded by the applications.* This means that if several threads of an executing application (or of independent applications running at the same time) request independent operating system services in parallel, then they must be serviced in parallel. Otherwise, the operating system becomes a bottleneck, limiting scalability and application speedup. Because we do not expect parallelism in servicing a single operating system request, and because an operating system is primarily demand driven, parallelism can only come from application demand. Therefore, the number of operating system service points must increase with the size of the system and the concurrency available in accessing the data structures must grow with the size of the system to make it possible for the overall throughput to increase proportionally.

Bounded overhead: *The overhead for each independent operating system service call must be bounded by a constant, independent of the number of processors* [4]. Otherwise, if the overhead of each service call increases with the number of processors, the system will ultimately saturate. This implies that the demand on any single resource cannot increase with the number of processors. For this reason, system wide ordered queues cannot be used and objects cannot be located by linear searches if the queue lengths or search lengths increase with the size of the system. Broadcasts cannot be used for the same reason.

Preserving locality: *The operating system must preserve the locality of the applications.* It is important to consider the memory access locality in large-scale systems, because for example, many large-scale shared memory multiprocessors have non-uniform memory access (NUMA) times, where the cost of accessing memory is a function of the distance between accessing processor and the target memory, and because cache consistency incurs more overhead in a large systems. Locality can be increased by properly choosing and placing data structures within the operating system, by directing requests from the application to nearby service points, and by enacting policies that increase locality in the applications' memory accesses. For example, policies should attempt to run the processes of a single application on processors close to each other, place memory pages in proximity to the processes accessing them, and direct file I/O to devices close by. Within the operating system, descriptors of processes that interact frequently should lie close together, and memory mapping information should lie close to the processors which must access them to handle page faults.

3 Clustering

We believe that an operating system designed using the structuring principles based on clustering and described in this section will automatically satisfy the above requirements for scalability, and is conducive for enacting appropriate policies that enhance locality, (although it should be noted that we do not discuss policy issues in this paper, but only describe the mechanisms that allow the scalable implementation of such policies).

3.1 Overview

The basic unit of structuring within Hurricane is a *cluster*, which consists of a symmetric micro-kernel, memory and device management subsystems, and a number of user-level system servers such as a scheduler and file servers. A cluster thus provides the functionality

of a complete, efficient small-scale symmetric multiprocessing operating system. Kernel data and control structures are shared by all processors within the cluster, giving good performance for fine-grained communication.

On larger systems, multiple clusters are instantiated so that each cluster manages a unique group of "neighboring" processing modules (including processors, memory and disks), where neighboring implies that in the absence of contention memory accesses within a cluster are never more expensive (and usually cheaper) than memory accesses to another cluster. Clusters cooperate and communicate to give users and applications an integrated and consistent view of a single large system.

The basic idea behind using clusters for structuring is to use multiple easy-to-design and hence efficient computing components to form a complete system. Clustering incorporates structuring principles from both tightly-coupled and distributed systems. By using the structuring principles of distributed systems, services are naturally replicated to distribute the demand, to avoid centralized bottlenecks, and to increase locality. The structuring principles of small-scale multiprocessors allow fine-grained data sharing for the common case where communication is local.

Despite the similarities between a clustered system and a distributed system, there are important differences that lead to completely different design trade-offs. For example, in a distributed system the hosts do not share physical memory, so the cost for communication between hosts is far larger than the cost of accessing local memory. In a (shared memory) clustered system, it is possible to directly access memory physically located in other clusters, and the costs for remote accesses are often not much higher than for local accesses. Moreover, demands on the system are different in the multiprocessor case, because of tighter coupling assumed by the applications.

3.2 Advantages

Clustering leads to a number of direct advantages. First, it provides a framework for managing locality. For the common case of small-scale parallel or sequential programs, all processes are scheduled onto the same cluster. This enhances performance as all interactions are local. Large-scale applications are scheduled across multiple clusters, and can benefit from the concurrency afforded through replicated system services; the components of the large-scale application typically request service from local servers.

Second, clustering allows performance tuning to different architectures. The appropriate cluster size for an architecture is affected by several factors, including the local-remote memory access ratio, the hardware cache size and coherence support, the topology of the interconnection backplane, etc. On hierarchical systems such as Cedar [15], Dash [17], KSR-1 [8], or Hector [20], a cluster might correspond to a hardware station. On a local-remote memory architecture, such as the Butterfly [5], a smaller cluster size (perhaps even a cluster per processor), may be more appropriate; in this case, clustering can be viewed as an extension of the fully distributed structuring sometimes used on these machines.

Finally, clustering simplifies lock structuring issues, and hence reduces code complexity, which can lead to improved performance and scalability. For example, Chaves reports that the fine-grained locking used in an unclustered system significantly increases the length of the critical path, even when there is no lock contention [9]. As well, deadlock can be a problem when several fine-grained locks must be held simultaneously. Because contention for a lock is primarily limited to the number of processors in a cluster, clustering allows for coarser grained locking, as we show in Section 7.

3.3 Challenges

Using clusters to structure an operating system for scalable systems also presents several challenges. One challenge is to completely hide from the applications the fact that the operating system is partitioned into clusters; to users and applications, the system should, by default, appear as a single, large, integrated system.[2]

Another challenge is to keep the complexity introduced by separating the system into clusters within reasonable bounds; clusters were introduced as a structuring mechanism primarily to reduce overall complexity. Therefore, the performance of a each cluster should be similar to that of a small-scale multiprocessor operating system. For those applications with a small degree of parallelism (or sequential applications) we expect the vast majority of all system interactions to be intra-cluster, and even for larger applications that span multiple clusters, we expect most of the system interactions to be intra-cluster, because of the replication of system services. Nevertheless, the overhead of inter-cluster communication should be minimized in order not to overly penalize those cases where inter-cluster communication is necessary.

A key issue in the design of a clustered system is the decision of how to distribute objects across clusters for increased locality and decreased contention, and once they are distributed how to locate them. It is also beneficial to replicate some objects for the same reasons, but they then have to be kept consistent. We describe examples of these design issues in Sections 5 and 6.

3.4 Research Issues

We are interested in determining how the size of the cluster affects performance, and how the workload and the attributes of the hardware affect the proper cluster size. There is a tradeoff between the advantages of replication and its overhead, as well as between the advantages of tight coupling and overhead of decreased locality.

We also wish to study whether each service (i.e., kernel, memory mangement, file service, etc.) is best served by the same cluster size, and whether one should entertain the possibility of having non-uniform cluster sizes, where the larger clusters are used to run applications with a higher degree of parallelism. In the long term, we intend to study policy issues for the various components of the system.

4 The Hurricane Operating System

In this section, we describe the basic organization of the Hurricane operating system, and how the components fit in with the clustered structure. Sections 5 and 6 then describe the clustered kernel and memory management in more detail.

The Hurricane kernel, which is similar in structure to the V kernel [10], provides for 1) address spaces, 2) processes, and 3) message passing. The address spaces are (initially empty) containers in which an arbitrary number of processes can run. Through the memory manager (and indirectly through various file servers), regions of the address space are bound to file regions before they can be used. Once a file is bound to a virtual address space, all references to memory are effectively references to the corresponding locations in the file, thus providing for a single-level store abstraction, where main memory is considered a cache of secondary store.

[2]Cluster can be made visible to sophisticated applications to allow performance optimizations.

Multiple processes can run within an address space. Processes within an address space typically communicate through shared memory, but messages are used to communicate between processes in different address spaces, and in particular between application and server processes.

Services not provided for by the kernel are provided for by servers. Some of these servers run in kernel space for protection (and sometimes for performance) reasons; examples are the device server that has to service interrupts, and the memory server that manages address spaces. The other servers run as normal user-level processes; they include the file server, the program manager, the pipe server, and the internet server.

4.1 The Hurricane Kernel

Processes are represented in the kernel by process descriptors, which are organized into various queues. Most operations on processes involve the dequeueing and enqueueing of process descriptors from and onto queues. For example, each processor in the cluster has a Ready Queue of processes ready to be executed, ordered by priority. A process is dispatched to a processor by dequeueing the descriptor at the head of its Ready Queue. If a processor's Ready Queue is empty, then for load balancing purposes a suitable process is taken from another Ready Queue in the cluster [12]. Other queues in the kernel include the Delay Queue for processes waiting for a timeout and a Receive-blocked Queue for processes that are receive-blocked waiting for a message. When a process sends a message to another process, then its descriptor (containing the message) is added to the Message Queue of the target process descriptor.

Within a cluster, all kernel data structures can be accessed (through shared memory) by any processor in the cluster. Three different mechanisms are made available for communicating across clusters: 1) direct shared memory access, 2) remote procedure calls (RPC), and 3) message passing. Cross cluster shared memory access is used for simple operations such as looking up the location of a process. Remote procedure calls are used for heavier weight, reasonably common operations where good performance is important; for example, RPCs are used to implement cross-cluster process migration. Finally, message passing is used for less-frequent, heavier-weight or non-time-critical operations, such as requests to the file server or requests to query or change the state of processes. Message passing is implemented using remote procedure calls, instead of vice versa as is usually the case in distributed systems. Generally, message passing is used to direct application requests to the kernel; if the local kernel cannot handle the request locally, it forwards the request to the appropriate cluster kernel.

Having the size of the clusters be smaller than the size of the system has three key advantages:

1. it localizes the kernel data structures that need to be accessed;
2. it reduces the number of process descriptors that must be managed within a cluster, thus reducing the average length of cluster-wide queues; and
3. it limits the amount of searching for ready processes when dispatching.

On the other hand, having a cluster span more than one processor reduces the amount of inter-cluster communication, and makes it easier to balance the load of the processors, leading to better system throughput and reduced application response time.

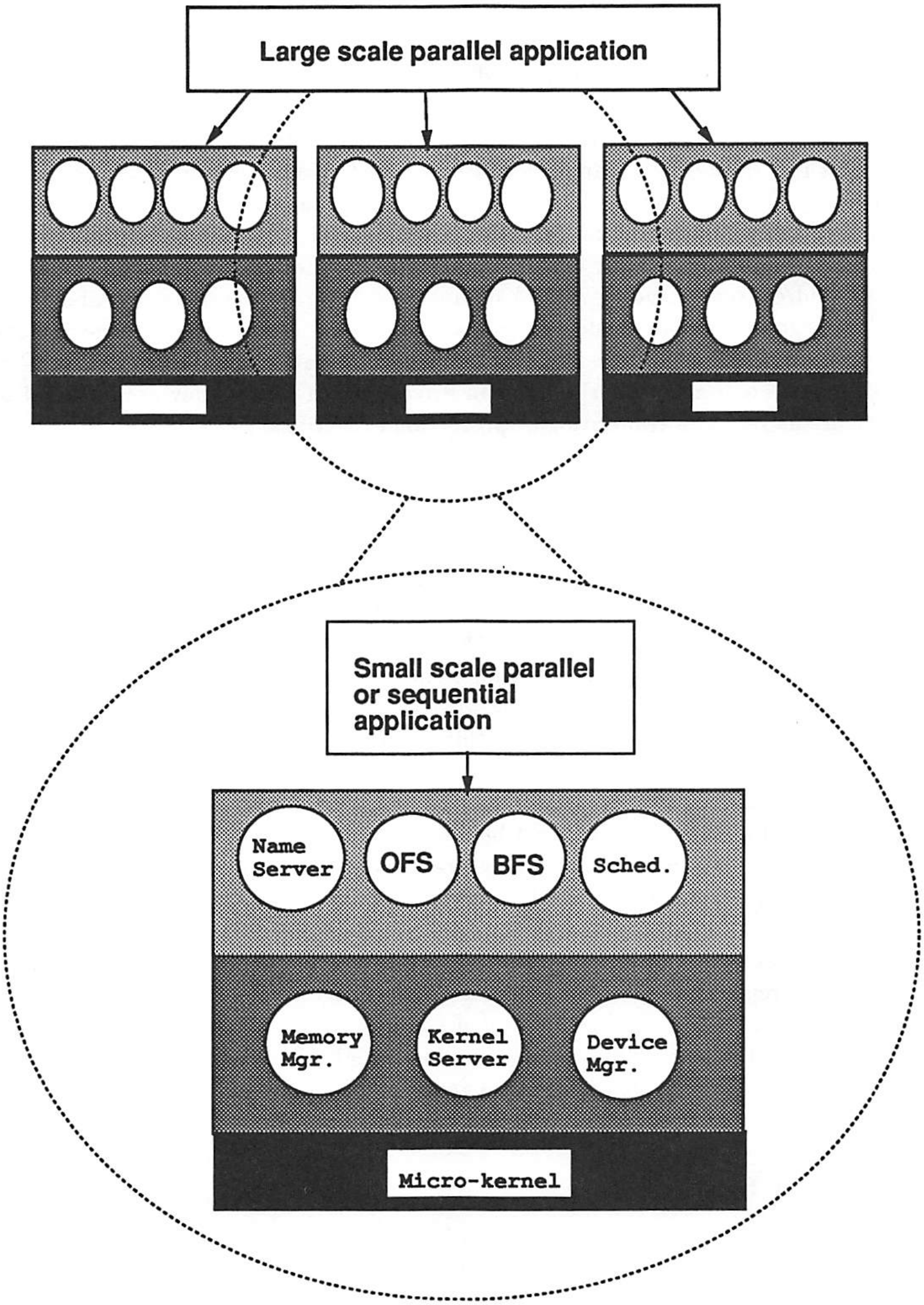


Figure 1: Clustering in the Hurricane Operating System.

4.2 Memory Management

Applications see memory as a three-tiered abstraction. The *address space* contains all the memory resources a program may access directly. Each address space is composed of a set of non-overlapping *regions*, which are contiguous sequences of *pages* and a set of attributes governing their management. The page is the lowest level of the hierarchy and is the smallest unit to which attributes and mappings can be applied. Each memory region is bound to a corresponding *file region*, so that accesses to memory behave as though they were accesses to the file directly. These virtual resources are managed by a *memory server* process on each cluster (see Figure 1). For example, it services requests from clients to allocate and deallocate address spaces and regions. A number of *copy processes* are also available to move larger data segments between address spaces. Finally, each cluster has a page-out daemon to support asynchronous writes. All memory-related operations at the physical level are on a per page basis, and are primarily demand-driven, which means they are initiated by the hardware as the result of translation or protection exceptions.

Clustering increases locality of accesses to the data structures of the memory manager, since each cluster maintains its own set of data structures to manage both the virtual and physical resources of local processes. This improves performance, since the structures that manage private data (such as process stacks) are local to the cluster on which the process executes. When applications share resources across clusters, the data structures that manage these resources can typically be replicated and cached locally, preventing bottlenecks and increasing concurrency. These structures and the mechanisms needed to keep them consistent are discussed in more detail in Section 6.

Clustering also provides a framework for enacting paging policies. At this time, the default policy is to share physical pages within a cluster, but to replicate and migrate pages across clusters. Several research groups have shown that page-level replication and migration policies can reduce access latency and contention for some applications [7, 11, 16]. But the overhead of these policies must be amortized to realize a net gain in performance. On machines where the local-remote access ratio is high, relatively few local accesses are sufficient to recoup these costs. However, technology advances have permitted this gap to narrow, and have allowed increases in hardware cache size. Both these trends mean that more local accesses are required to justify a page movement, which argues for less aggressive placement/replication policies. Moreover, replication lowers the effective utilization of memory by increasing the resident set size of an application, which could lead to increased disk paging when the level of multi-programming is high. The current default policy therefore limits the amount of replication or migration, reducing overhead, but still allows for the reduced latency and increased concurrency of localized accesses to replicated data.

4.3 The File System

The Hurricane file system differs from that of many other operating systems in that much of its functionality is implemented by user-level I/O libraries. The *Alloc Stream* library [14], used by all applications, implements `read` and `write` operations by mapping file regions into the application's address space and copying the data to or from these mapped regions (staying entirely within the application level).

There are three user-level servers in the file system: the *Name Server*, the *Open File Server* (OFS), and the *Block File Server* (BFS). The name server manages the Hurricane name space; it is the portion of the file system that understands directories and symbolic

links. The name server also manages all accesses to logical file information, such as file length, modification times, etc.. The open file server maintains the file system state kept for each open file, and uses this information for translating application requests into operations to the memory manager. Finally, the BFS controls the local system disks. It is responsible for directing operations to the device drivers associated with the target disk.

The name server, OFS, and BFS are all replicated on a per cluster basis, and the Alloc Stream library always directs application requests to the local servers. Moreover, the memory manager also directs the disk I/O operations to the local BFS, who in tern directs these operations to the other BFS's if necessary. In order to keep the view of the file system consistent across clusters, the servers communicate (using message passing) with the corresponding servers on other clusters.

The state of the name servers are replicated on a need-to-know basis, and an updating mechanism (instead of invalidating) is used for consistency This strategy is appropriate under the assumption we make that 1) there is a large degree of locality in the directories an application accesses, and 2) directories are either accessed by a a large number of applications in a read only fashion, or a small number of applications in a read/write fashion.

Requests to an open file are always directed to the local OFS. In two cases the open state may become used at other clusters, namely: 1) the process that opened the file passes the file handle as a capability to another program, or 2) several processes of a program spanning multiple clusters are accessing the file. The open file state is replicated on demand from the OFS that opened the file originally (and the id of which is encoded in the file handle). Clustering therefore distributes the overhead of file I/O across the clusters (for concurrency) in a natural way.

For those operations that require I/O, we believe clustering can provide a framework for balancing the load across the disks of the system. Although we have not yet implemented this part of the file system we believe it would be beneficial if the blocks of individual files could be distributed and replicated across a number of disks. All requests for disk I/O from a particular cluster would still be directed to the local BFS, but the BFS would forward the request to the appropriate block file server on another cluster if it cannot handle the request locally.

4.4 Scheduling

The primary purpose of the scheduling subsystem is to keep the loads on the processors (and possibly other resources, such as memory) well balanced, in order to decrease the average response time of the applications. In Hurricane we expect this to occur at two levels. Within a cluster, the load between the processors is balanced at a fine granularity through the dispatcher (in the micro-kernel). Higher-level scheduling servers then balance the load across clusters at a coarser granularity, by assigning newly created processes to specific clusters, and by migrating existing processes to other clusters.

Although we have not yet implemented a higher-level scheduler, motivation for this approach stems from simulation studies by Zhou and Brecht [21], where they show that clustering can noticeably improve overall performance. In a clustered system, the processes of an application are scheduled to run in a single cluster, unless there are performance advantages for a job to span multiple clusters. Hence, for parallel programs with a small number of processes, all of the processes will run on the same cluster. For larger-scale parallel programs that span multiple clusters, the number of clusters spanned is minimized.

A similar structuring mechanism for scheduling has been proposed by Feitelson and Rudolph [13], and by Ahmad and Ghafoor [2].

5 The Hurricane Kernel and Cross-Cluster Communication

The kernels of each cluster in the system communicate and cooperate in order to provide the processes and users a consistent view of the system. While shared memory is used as the primary mode of communication within a cluster, three different mechanisms can be used for the kernels to communicate across clusters.

First, shared memory access remains a viable option for cross cluster communication, particularly for light-weight operations that make only a few references. For example, shared memory is used to locate the current position of a process descriptor. When a process is created, it is assigned a process identifier within which the id of the *home* cluster is encoded. Usually, a process remains within its home cluster, but if it migrates, then a record of its new location is maintained at the home cluster. As a process migrates from cluster to cluster, its location information at the home cluster is updated each time. Remote shared memory access to this data structure is appropriate, because this information is maintained in a hash table only a small number of references are needed for lookup.

Remote procedure calling is a second mechanism for cross-cluster communication. It is implemented using remote interrupts (and is for this reason referred to by Chaves [9] as "bottom half invocation"); the interrupted cluster obtains the call information and arguments through shared memory directly. The use of remote procedure calls allow data accesses to be local to the interrupted cluster, but the cost of the interrupt (i.e. the cycles that are stolen and the register saving and restoring) must be amortized.

Finally, message passing can be used to communicate between processes on different clusters. For message passing within a cluster, the message is copied into the descriptor of the sending process, which in turn is queued in the message queue of the target process descriptor. For message passing across clusters, remote procedure calls are used to interrupt the target cluster to 1) allocate a message retainer (similar to a process descriptor), 2) copy the relevant information including the message by directly accessing the source processor descriptor on the source cluster, and 3) add the message retainer to the message queue of the target descriptor. The actions between the source and the target cluster are similar to the actions taken by the V kernels on different hosts, except for the fact that data is passed through shared memory directly instead of through a network packet. Other message passing primitives, such as Reply and Forward, are implemented in a similar fashion. Note that by using local message retainers as opposed to the source process descriptors which are remote, the message queue will always be an entirely local data structure.

Remote procedure calls are used to implement cross-cluster process migration, by allocating a new process descriptor at a target cluster and copying the contents of the previous descriptor into the new one. This is done by the target cluster in response to a remote procedure call. When the migrated process begins to execute in its new cluster, it will immediately page fault, at which time the memory structures needed by the processes are (re)constructed, as described in Section 6.

Application-level processes use message passing to communicate with a kernel-level server that runs in each cluster to service non-time-critical kernel requests on behalf of remote clients. For example, if a client wishes to query the priority of a process, then the query is first directed to the local cluster. Once the local kernel determines that the target

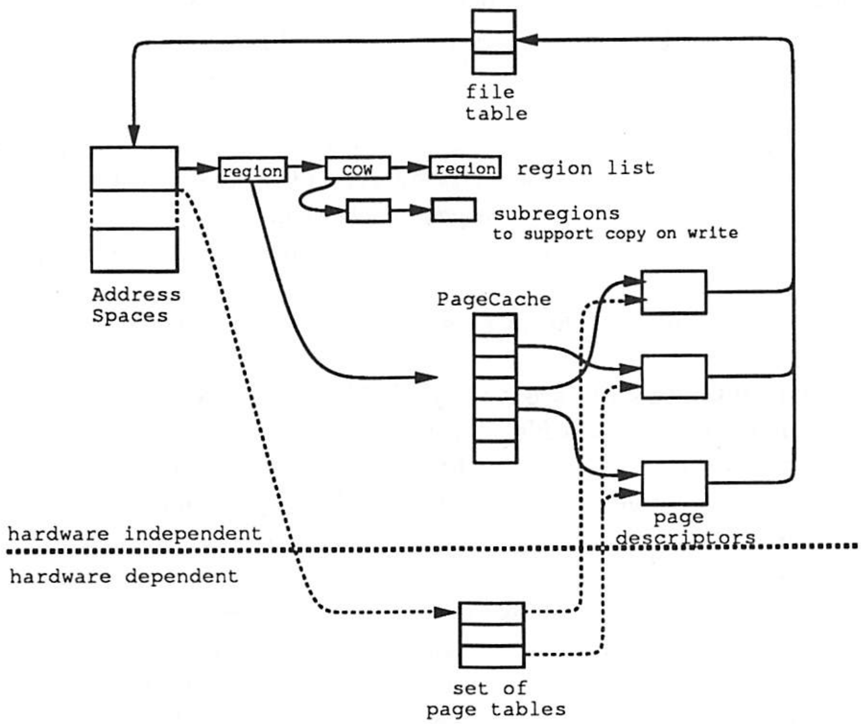


Figure 2: Per-cluster data structures.

process is not local, the current location of the target process is determined (using shared memory directly, as described above), and a request message is forwarded from the client to the kernel on the target cluster. Using message passing for these operations greatly simplifies the design of the cluster kernel.

Message passing is the only form of communication available for user-level servers that need to be accessed remotely (assuming they do not map in a common file so that they can share memory directly). However, other kernel-level services, such as the memory manager described in the next section use all three mechanisms: shared memory, remote procedure calling, and message passing.

6 Memory Manager Implementation

To handle a page fault, the memory manager first determines the address space of the faulting process, and then determines the memory region of the address space identified by the faulting virtual address.[3] If it exists, the memory region corresponds to a file region. The file region and the offset of the virtual address within the memory region together identify a particular file block. The memory manager then determines whether that page is already cached locally. If so, it maps the page for the faulting process, by updating the page table (assuming access permissions allow this). Otherwise, it allocates a new page frame for it, and either forwards a request to the Block File Server for DMA, or zeroes it.

Figure 2 shows the most important data structures needed to manage both the virtual and physical resources within a cluster. All processes of a single program share a single address space descriptor (ASID) that stores program-wide information. Each address space

[3]The regions are created in response to explicit application requests to the memory manager.

has a list[4] of bound *Region* records, which maintain the virtual limits of each region, the file region the memory region is bound to, and the mapping and placement attributes of the region. (Sub-region lists are used to identify pages not yet copied from regions marked copy-on-write.)

Each physical page in the cluster has a *page descriptor* associated with it. This structure identifies the physical address of the page, its state, and the file block stored there. All pages that contains valid file data are kept in a *page cache*, which is a hash table indexed by the file block, and is used to determine whether a file block is already resident in the cluster [3]. When the page is no longer referenced by any process in the cluster, it is placed on a free list, but remains in the page cache in case the file block is referenced again in the near future.

The last important data structure is the *file table*, which is a hash table of *FileReference* structures with one entry for each file with a region mapped into an address space. Each entry maintains a set of back-pointers to all regions that map some portion of the file. These pointers are used, for example, after selecting a page for page-out to unmap the file block from all processes that map the page.

In summary, the virtual memory resources are kept on a per address space basis, while physical resources are kept on a per page basis. Files form the logical interface between the two levels, since regions are bound to files and pages cache file data. For a page fault, first the address space descriptor and then the region descriptor is used to identify a file block bound to the faulting address. The page cache is used to get the physical address of the page containing the file block. The reverse path, from the page through the file table leads to the address spaces that map a particular file page.

We now describe how these structures are maintained across clusters, and the additional data structures needed to maintain global page location and coherence. Because the per address space structures are referenced on every page fault, it is important to keep the cost of accessing this information to a minimum, even when the processes of a program span multiple clusters. Consequently, each cluster keeps a local "working set" of per address space data structures that represent the virtual resources used by the processes on that cluster. To maintain the integrity of these local structures, all modifications are directed through the *home cluster*, which serves as a point of synchronization. Currently, the home cluster is designated as the cluster on which the address space was created. The address space structures are accessed read-only far more often then they are modified, so it is acceptable to increase the overhead of modifications, if in turn the read-only access overhead is decreased.

All application requests are initially directed to the local memory server process of Figure 1. If the request cannot be handled locally, it is forwarded to the manager on the home cluster for handling. For consistency, the home cluster propagates any changes to the address space data structures to all clusters which have copies of the structures being modified. For this purpose, a *cluster set* bit vector is kept in the `ASID` structure on the home cluster. The cluster set identifies the other clusters which may have a copy of the structures and is used to limit the extent of the broadcast.

The address space descriptor is replicated from the home cluster to a new cluster when a newly migrated process first accesses memory at the new cluster. This replicated address space is initially empty; the region list (and hardware dependent page tables) are built up from the home cluster by *lazy replication* as the program executes when a region descriptor cannot be found locally. The sub-region lists, which are used by copy-on-write regions, are

[4] In actuality, this list is implemented as a balanced binary search tree for fast concurrent access.

not replicated and reside on the home cluster only. This is because copy-on-write regions are mostly used for the initialized data section of programs, and therefore tend to be short and short-lived. By not replicating the sub-region lists, the number of remote searches is increased, but we believe it will be cheaper than repeatedly replicating and modifying the sub-regions, since a modify requires a broadcast update by the home cluster. The page descriptors are also replicated, because it reduces the number of remote operations, and because it is convenient for the implementation of some of the paging policies.

The page cache described earlier is used to search for file blocks cached in the memory of the cluster. With clustering it is now necessary to be able to locate file blocks in other clusters, since it is typically less expensive to copy a page from a remote cluster than it is to copy it in from disk. This is complicated by the fact that pages may be replicated across clusters, so that a search may be required to return the "closest" copy. Our current implementation uses a full directory scheme[1] for this purpose.

The directory scheme is conceptually simple: each valid file block has an entry associated with it that keeps a bit vector and some state. Each bit of the bit vector, called the *copy set*, represents a cluster that may have a copy of the data. A directory entry is also used as a point of synchronization, needed to maintain integrity on the shared structures (when adding or deleting entries), and to keep the file blocks consistent across memory and with respect to disk. For example, accesses to a file block must be suspended while it is in transit from backing store, and modifications must be delayed if the block is currently being replicated. The directory is physically distributed across all clusters in the system. This is necessary because the number of directory entries is proportional to the size of main memory. Moreover, it also allows concurrent searching and updates to independent file blocks. The directories are accessed through remote shared memory.

7 Experimental Results

In this section, we present the results of simple experiments and performance measurements.

7.1 Experimental Environment

All our experiments were conducted on a 16 processor Hector shared-memory multiprocessor running the Hurricane operating system. Hector is a hierarchical multiprocessor designed for scalability [20], where a number of processing modules are connected by bus to form a so called *station*, which in turn are connected by a hierarchy of rings. The processing modules of the current implementation contain a Motorola 88000 processor, 16 Kbytes instruction cache, 16 Kbytes data cache and 4 Mbytes RAM.[5] The particular configuration used in our experiments consists of 4 processing modules per station and 4 stations connected by a ring. The prototype, which has been running for close to 2 years, runs at 20 MHz.

Hector is a NUMA system in that the memory access times differ depending on the distance between accessing processor and target memory, although the difference on Hector is typically less pronounced than on other NUMA systems, such as the TC-2000. In our configuration, it takes 20 cycles to fill a 16 byte cache line from local (on-board) memory, 26 cycles if filled from off-board but on-station memory, and 30 cycle if filled from the memory of another station.

[5]The hardware in the current implementation can accommodate up to 128 Kbytes of instruction cache 128 Kbytes data cache and 16 Mbytes RAM.

All experiments were run on a fully configured version of Hurricane with all servers running, but no other applications were running at the time. We have spent considerable efforts to optimize the memory management for improved performance in general, and maximum concurrency (even within a cluster) in particular. In contrast, the rest of the kernel has not been optimized and is rather crude; for example, only a single lock is available to control concurrency (effectively serializing kernel operations). To reduce caching effects and obtain fairer performance numbers, we disabled the caching of all data within the kernel except for stack data; instructions were cachable.

7.2 Basic Operations

To assess the overhead introduced by clustering, it is interesting to compare the cost of intra-cluster and cross-cluster operations. The cost of handling a basic read page fault entirely within a cluster (of size 4) from the time of the trap until the time control is returned to the faulting (user-level) process is 128 microseconds. This includes, the lookup of the address space descriptor, the region descriptor, finding the page in the page cache, and mapping it in the processes page table. The corresponding time to handle a page fault for a page that is located in another cluster is 243. microseconds The extra overhead is due to the directory lookup that is now necessary, and the overhead of replicating the page descriptor.

The cost of message passing increases from 384 microseconds within a cluster to 584 microseconds across clusters. This is the cost of a 32 byte send-response message transaction, from a (user-level) process in one address space to a process in another address space and back again. The extra overhead in the remote case is for the two remote procedure calls needed for the sending and the replying of the message, each of which involve remote processor interrupts (together with saving and restoring the registers of the interrupted processor), the allocation of the remote message retainer at the sender side, and the cross-cluster copying of the request and reply messages. The cost for a round-trip cross-cluster null-RPC is 60 microseconds.

7.3 Synthetic Stress Tests

We ran several tests to study the effects of clustering on throughput and response time. In these tests n processes run on different processors, where n varies from 1 to 16. In the first test, the processes page fault as quickly as possible. Each process in a loop (1) binds a region of 128 pages in to their address space, (2) causes a fault for each page by reading a word from it, (3) synchronizes with the other participating processes through a barrier, and (4) unmaps the region. The pages when first touched cause a fault, because they are not yet mapped into the processes address space, but a cached copy of the page can always be found in local memory so no I/O is required to service the fault.

Figure 3.a shows the response time per page fault for four different configurations of Hurricane:[6] and Figure 3.b shows the corresponding normalized throughput:

Config-1: 1 cluster of 16 processors,
Config-2: 2 clusters of 8 processors,
Config-4: 4 clusters of 4 processors, and
Config-8: 8 clusters of 2 processors each.

[6] Note that the hardware configuration does not change.

Two variants of the test were run. In the first variant (represented by solid lines in the figure), the n processes of the test are spread as evenly across the clusters as possible. For example, with Config-4 and $1 \leq n \leq 4$, the processes are assigned to different clusters. The resulting response times are the same for these four tests, which is the expected behavior, because no cross-cluster interaction is needed to service the page faults. For the same reason, we obtain throughout the same response times at equal levels of cluster loading in all of our tests, indicating that (for a system of this size) the speedup is linear in the number of clusters, regardless of size.

As more processes begin to run within a cluster, response time begins to increase. This increase is entirely due to contention: memory contention to a small degree and lock contention to a large degree. The curve for Config-1 with 16 processors in a cluster is only shown up to 8 processes; for 16 processes the response time was measured to be about 1300 microseconds. Note that when there is not much contention the response times get marginally better as we go to smaller clusters. This is due to the higher degree of locality: in a 4 processor cluster, the probability of any memory manager data structure being on the local processing modules is 1/4, while it is about 1/2 for 2 processor clusters.

In the second variant of this first test (represented by dashed lines in the figure), processes are first added to one cluster until each processor in the cluster is assigned a process, before beginning to assign processes to new clusters. Since Config-1 has only one cluster, the curve of variation 1 applies. For Config-2 with 2 clusters of 8 processors each, we find that the response time increases up to 8 processes similar to the Config-1 case. But with 9 processes, the average response time is lower, because the one process running alone in the second cluster is much faster decreasing the overall average. At 16 processes, the response time is then again as high as for 8 processes. Similar effects can be seen for the other configurations as well. For example in Config-4, the response time peaks at 4, 8 12 and 16 processes.

In the second test, shown in Figure 3.c, the participating processes in a loop request from the kernel the id of their creator; this occurs by sending a message to the cluster-local kernel server and obtaining a response. The same configurations and the same variations of the test as described above were used. The exact same behavior can be observed: the throughput increases linearly with the number of clusters, and the response times of the second variation of the tests bounces with peaks when the number of processes is evenly divisible by the cluster size.

The third test, shown in Figure 3.d, exhibits again the same behavior, this time when measuring message passing times. The dotted line in this figure shows the times for a configuration with 16 clusters of one processor each, where each Send-Reply transaction involves 2 RPCs.

It is interesting that all three experiments behave the same way despite the fact that the first test exercises an optimized portion of the system, while the second and third test exercise unoptimized portions of the system. This is evident when comparing response times (which are much lower for the page fault test), but also when comparing the shape of the solid curves for small number of processes: in the first test, the curve is flat at first, but then increases linearly,[7] whereas in the second and third tests the curves increase linearly from the start, because of the single lock.

In all of the above tests, the operations do not require cross-cluster communication,

[7] In the first test, the contention is caused by a common address space descriptor must be accessed for all processes.

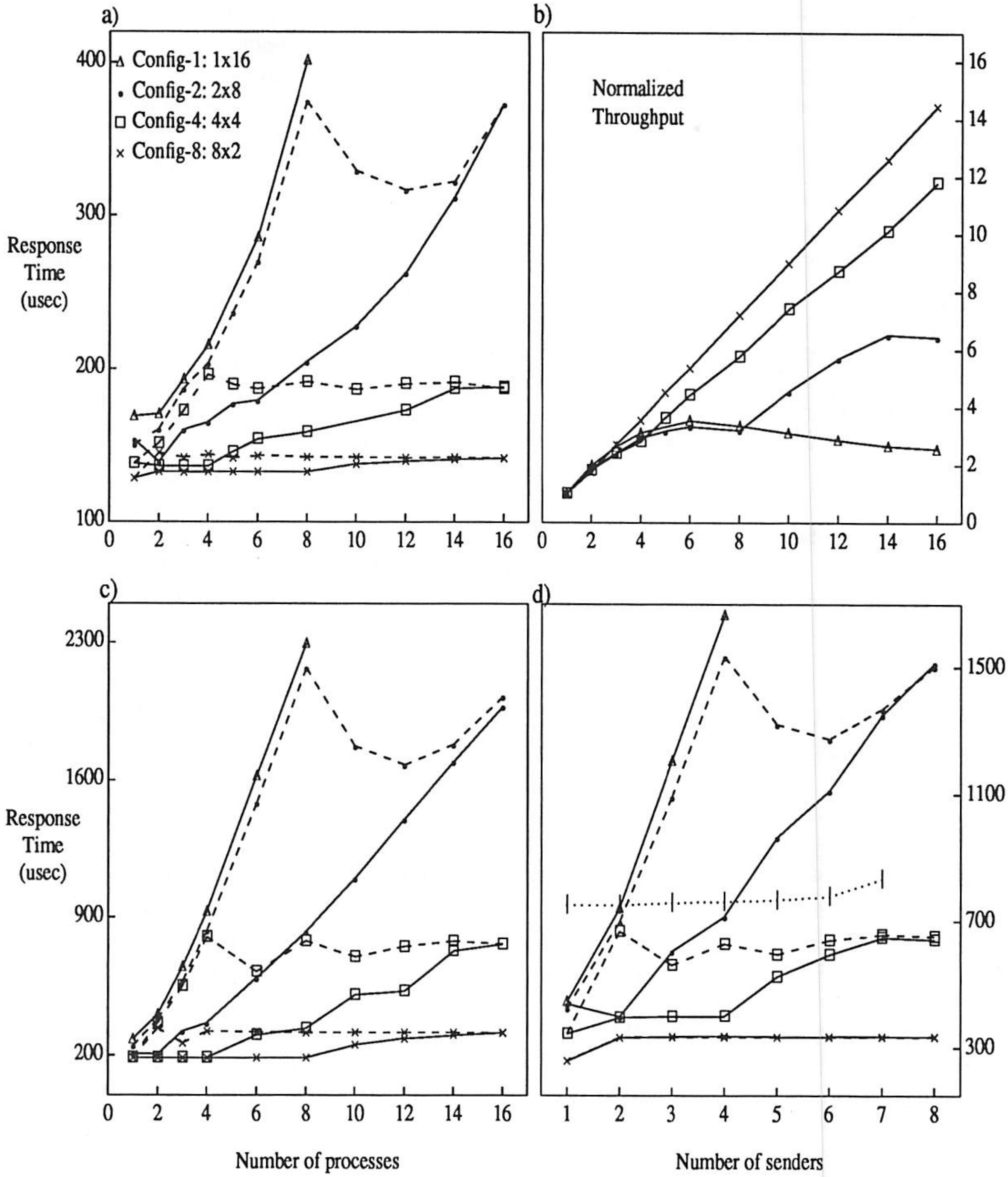


Figure 3: The response times for (a) page fault handling, (c) kernel request servicing, and (d) message passing for various cluster configurations and loads. (b) depicts the corresponding throughput for page fault handling.

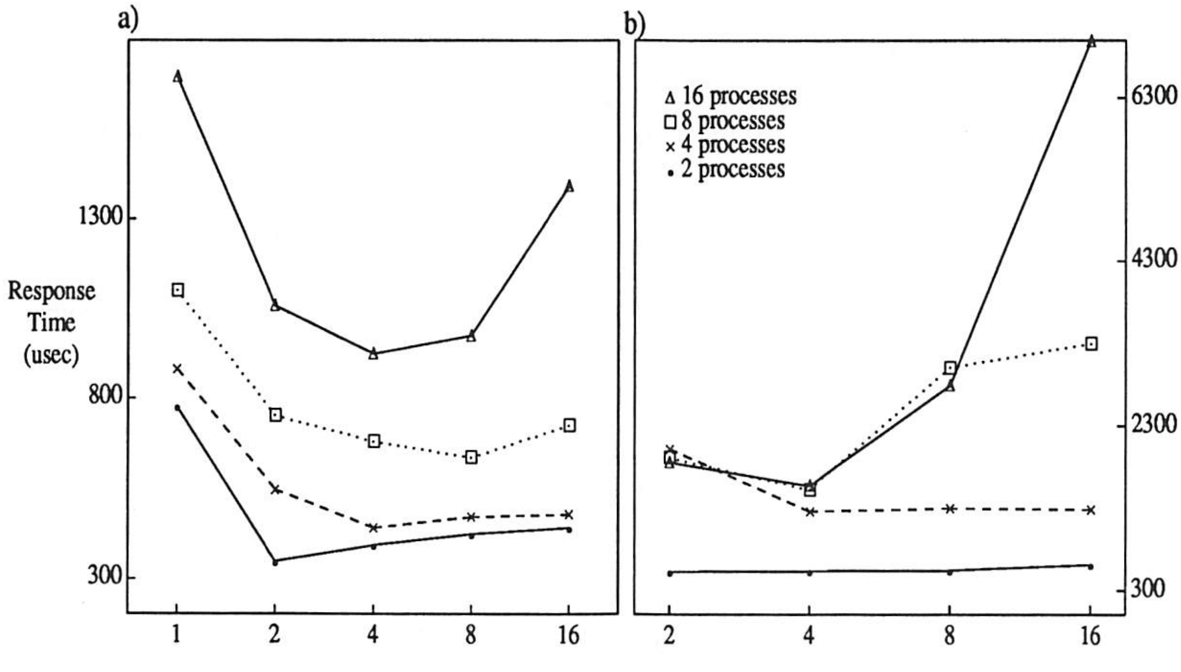


Figure 4: The response times for (a) page fault handling, and (b) message passing requiring cross-cluster communication, for varying cluster sizes. The different curves represent the tests performed with different numbers of processes.

in order to isolate the effects of locking. The next two tests involve a significant amount of cross-cluster interaction. In Test 4, one process initializes a number of pages, after which n processes start accessing these pages, causing them to fault. The n processes are assigned one to a processor while minimizing the number of clusters they span. Figure 4.a shows the response times for a page fault for cluster sizes of 1, 2, 4, 8 and 16. From the figure, one can see that when 2. 4 or 8 processes participate, the ideal cluster size is 2, 4 and 8, respectively. This is because all faults can be handled locally with no cross-cluster communication. Note, however, that when 16 processes fault on these pages, a cluster size of 4 yields the best response time; contention within the cluster increases with the cluster size, and at this high degree of sharing, the overhead of contention dominates the overhead of remote operations.

In Test 5, shown in Figure 4.b, n sender and receiver process pairs are run, each pair on its own processor. The sender process on processor i always communicates with the receiver process on processor $(i+1) mod\ n$. The figure shows that with small cluster sizes, the overhead of cross-cluster communication dominates over the overhead for lock contention, while for larger cluster sizes, lock contention overhead dominates over overhead for cross-cluster communication. For this particular test, the best cluster size is 4 across all loads measured.

8 Concluding Remarks

We have introduced the concept of clustering as a way to structure operating systems for scalability, and described how we applied this structuring technique in our experimental operating system. We presented performance results from our prototype that demonstrate

the characteristics and behavior of the clustered system. In particular, we showed how clustering trades off the efficiencies of tight coupling for the advantages of replication, increased locality and reduced lock contention but at the cost of cross-cluster overhead.

The system we have designed and implemented meets a number of requirements necessary for scalability. First, we showed that independent operations directed to different clusters can be serviced completely in parallel; there are no central servers, or system-wide locks. The system can therefore preserve the parallelism afforded by the applications.

Second, the number of queues and the number of service points in the system increases with the size of the system. The overhead for each independent operation is therefore independent of the size of the system.

Finally, the system preserves the locality of the applications. Service requests are directed to local service points, and state is distributed and replicated in order to localize access whenever possible. With only one exception, the data structures of only those clusters directly involved in a service request are accessed when servicing the request. The exception involves lookup tables accessed by hash functions. These tables are used to locate objects (such as process descriptors, or cached file pages), can be accessed concurrently, and span the system to distribute the load.

Our work on clustering and Hurricane is still in its initial stages. We intend to pursue further experiments to determine how cluster size affects performance, and how the workload and the attributes of the hardware affect the ideal cluster size. We also wish to study whether each service (i.e., kernel, memory mangement, file service, etc.) is best served by the same cluster size, and whether one should entertain the possibility of having non-uniform cluster sizes, where the larger clusters are used to run applications with a higher degree of parallelism. In the long term, we intend to study policy issues for the various components of the system, including paging, scheduling and I/O subsystems.

Acknowledgements

We wish to thank David Blythe, Yonatan Hanna, and Songnian Zhou for their significant contribution to the design and implementation of Hurricane. We also greatfully acknowledge the help of Tim Brecht, Ben Gamsa, Jan Medved, Fernando Nasser, and S. Umakanthan.

References

[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proc. 15th International Symposium on Computer Architecture*, pages 280–289, Honolulu, Hawaii, May 1988.

[2] I. Ahmad and A. Ghafoor. Semi-distributed load balancing for massively parallel multicomputer systems. *IEEE Transactions on Software Engineering*, 17(10):987–1004, October 1991.

[3] M.J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, 1986.

[4] A. Barak and Y. Kornatzky. Design principles of operating systems for large scale multicomputers. Technical Report RC 13220 (#59114), IBM T.J. Watson Research Center, 10 1987.

[5] BBN Advanced Computers, Inc. *Overview of the Butterfly GP1000*, 1988.

[6] BBN Advanced Computers, Inc. *TC2000 Technical Product Summary*, 1989.

[7] W.J. Bolosky, R.P. Fitzgerald, and M.L. Scott. Simple but effective techniques for NUMA memory management. In *Proc. 12th ACM Symp. on Operating System Principles*, pages 19–31, 1989.

[8] H. Burkhardt. *KSR1 computer system.* Comp.arch Netnews article, Kendall Square Research, February 1992.

[9] E. Chaves, T. J. LeBlanc, B. D. Marsh, and M. L. Scott. Kernel-kernel communication in a shared-memory multiprocessor. In *Proc. Second Symposium on Distributed and Multiprocessor Systems*, pages 105–116, Atlanta, Georgia, March 1991. Usenix.

[10] David R. Cheriton. "The V Distributed System". *Communications of the ACM*, 31(3):314–333, March 1988.

[11] A.L. Cox and R.J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINUM. In *Proc. 12th ACM Symp. on Operating System Principles*, pages 32–44, 1989.

[12] S. Curran and M. Stumm. Scheduling a Unix workload on small-scale, shared memory multiprocessors. *Computer Systems*, 3(4):551–579, October 1990.

[13] D.G. Feitelson and L. Rudolph. Distributed hierarchical control for parallel processing. *Computer*, 23(5):65–81, May 1990.

[14] O. Krieger, M. Stumm, and R. Unrau. Exploiting the advantages of mapped files for stream I/O. In *Proc. 1992 Winter USENIX Conf.*, 1992.

[15] D.J. Kuck, E.S. Davidson, D.H. Lawrie, and A.H. Sameh. Parallel suppercomputing today and the Cedar approach. *Science*, pages 967–974, 1986.

[16] R.P. LaRowe Jr., C.S. Ellis, and L.S. Kaplan. Tuning NUMA memory management for applications and architectures. In *Proc. 13th ACM Symp. on Operating System Principles*, October 1991.

[17] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *17th Intl. Symp. on Computer Architecture*, 1990.

[18] D. Nussbaum and A. Agarwal. Scalability of parallel machines. *Communications of the ACM*, 34(3):56–61, March 1991.

[19] G.F. Pister, W.C. Brantley, and George D.A. The IBM parallel research processor prototype. In *Proc. Intl. Conf. on Parallel Processing*, pages 764–769, 1985.

[20] Z.G. Vranesic, M. Stumm, D. Lewis, and R. White. Hector: A hierarchically structured shared-memory multiprocessor. *IEEE Computer*, 24(1), January 1991.

[21] S. Zhou and T. Brecht. Processor pool-based scheduling for large-scale NUMA multiprocessors. In *Proc. ACM Sigmetrics Conference*, September 1990.

The USENIX Association

USENIX, the UNIX and Advanced Computing Systems professional and technical organization, is a not-for-profit membership association made up of systems researchers and developers, systems administrators, programmers, support staff, application developers and educators.

USENIX is dedicated to:

* fostering innovation and communicating research and technological developments,
* sharing ideas and experience, relevant to UNIX, UNIX-related and advanced computing systems
* providing a forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, the Association sponsors two annual technical conferences and frequent symposia and workshops addressing special interest topics, such as C++, distributed systems, Mach, systems administration, and security. USENIX publishes proceedings of its meetings, a bi-monthly newsletter ;login:, and a refereed technical quarterly, Computing Systems. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts.

There are four classes of membership in the Association: student, individual, institutional, and supporting, differentiated primarily by the fees paid and services provided. The supporting members of the USENIX Association are:

Digital Equipment Corporation
Frame Technology, Inc.
Matsushita Graphic Communication Systems, Inc.
mt Xinu
Open Software Foundation
Quality Micro Systems
Rational Corporation
Sun Microsystems, Inc.
Sybase, Inc.
UNIX System Laboratories, Inc.
UUNET Technologies, Inc.

For further information about membership or to order publications, contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710-2565
Telephone: 510/528-8649
Email: office@usenix.org
Fax: 510/548-5738

ISBN 1-880446-42-1